Testing Intermediate Representations for Binary Analysis

Soomin Kim*, Markus Faerevaag*, Minkyu Jung*, SeungIl Jung*, DongYeop Oh*, JongHyup Lee†, Sang Kil Cha*

*KAIST, Republic of Korea
{soomink, mfaerevaag, hestati, sijung, oh51dy, sangkilc}@kaist.ac.kr

†Gachon University, Republic of Korea
jonghyup@gachon.ac.kr

***Abstract*—Binary lifting, which is to translate a binary executable to a high-level intermediate representation, is a primary step in binary analysis. Despite its importance, there are only few existing approaches to testing the correctness of binary lifters. Furthermore, the existing approaches suffer from low test coverage, because they largely depend on random test case generation. In this paper, we present the design and implementation of the first systematic approach to testing binary lifters. We have evaluated the proposed system on 3 state-of-the-art binary lifters, and found 24 previously unknown semantic bugs. Our result demonstrates that writing a precise binary lifter is extremely difficult even for those heavily tested projects.**

I. Introduction

Understanding binary code is crucial in software engineering and security research. Source code is not available when it comes to Commercial Off-The-Shelf (COTS) software, malware, or legacy code. Even if we have access to the source code, we cannot trust it when the compiler is not in the trusted computing base [54]. Last but not least, even a trusted compiler can produce binary code that is semantically different from the source code [10].

In the past few decades, there has been much research on binary code analysis and its applications including binary instrumentation [13], [36], [40], [46], binary translation [20], software hardening [16], [26], [59], [60], software testing [9], [17], [27], CPU emulation [12], [42], malware detection [18], [33], automated reverse engineering [21], [38], [39], [52], [57], and automatic exploit generation [15].

The very first step in binary analysis is to convert a binary executable into an Intermediate Representation (IR), which precisely represents the operational semantics of the binary code. Such a process is often referred to as binary lifting, and nearly all of the above approaches involve binary lifting either explicitly or implicitly. The converted IR is the basis for any binary analysis techniques. Therefore, any bug in the resulting IR can immediately invalidate the binary analysis results. A taint-based malware detection system [58] can report false alarms. Instrumented programs can fail or even crash. For example, a single IR bug in QEMU indeed resulted in a failure of the entire system emulation [5].

Unfortunately, engineering a precise binary lifter is notoriously difficult. First, the volume of an instruction set manual is too large to comprehend. For example, the manuals for Intel [2] and ARM [1] currently consist of 4,700 and 6,354 pages, respectively, at the time of writing. Even worse, the number keeps increasing as new CPU features are introduced. Furthermore, the semantics of CPU instructions are often informally (and vaguely) defined in natural language. Finally, there are undefined or undocumented semantics that are implemented on a real CPU. Thus, developers often write IRs on a trial-and-error basis, which is error-prone.

Despite these issues, there has been surprisingly little effort on testing the correctness of binary lifters. The most relevant work to date is that of Martignoni *et al.* [43], [44], which attempted to leverage differential testing on QEMU [12] and Bochs [37]. Particularly, they compared the state between a physical and an emulated CPU after executing randomly chosen instructions on both to discover any semantic deviations.

Although their technique can be applied to testing binary lifters, it is fundamentally limited because its effectiveness largely depends on randomly generated test cases. Typically, semantic bugs in binary lifters are triggered only with specific operand values. Therefore, a random test case generation does not help much in finding such bugs. For example, the `bsf` instruction of x86 searches the source operand for a least significant *set bit* within a while loop that checks one bit at a time. There are 32 different numbers of loop iterations possible, but it is unlikely that randomly generated test cases would cover all such cases. Let us suppose there exists a semantic bug in an IR of `bsf` instruction when the maximum number of iterations (= 31) is reached. This condition can occur only when the source value is 2^{31}, and the probability of generating the buggy operand value at random is $1/2^{32}$.

The key question that motivated our research is: can we test binary lifters without relying on randomly generated test cases? One potential approach is to use traditional formal verification [31]. Particularly, we can build a symbolic formula that encodes all execution paths of an IR instance lifted from a single machine instruction. We then check if the symbolic formula matches the formal specification of the instruction.

978-1-5386-2684-9/17 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research

Although this is a totally plausible approach, there is no such formal specification available for modern CPUs.

In this paper, we propose a novel approach to finding semantic bugs in binary lifters, called N-version IR testing. Our approach is inspired by N-version disassembly [47], where the outputs of independently developed disassemblers are compared to each other for a given input binary in an attempt to find bugs in disassemblers. Unlike N-version disassembly, however, we do not syntactically compare the outputs of the lifters under test. Instead, we formally check the semantic equivalence between N distinct IRs obtained from a single binary instruction. Therefore, any semantic discrepancy means that there is at least one semantic bug on the lifters under test. Furthermore, our approach does not rely on random test cases as in the previous approaches.

MeanDiff implements this idea to find semantic bugs on existing binary lifters. Applying it to 3 existing binary lifters found 24 previously unknown semantic bugs, which were all manually confirmed, and reported to the developers. Our experience shows that buggy IRs are widely used in existing binary analysis techniques and tools, and can affect their precision. Although our current implementation is specific to x86 and x86-64, the proposed technique is general and can be applied to other instruction set architectures.

Overall, this paper makes the following contributions:

1) We systematically study the characteristics of existing intermediate representations generated from open-sourced binary lifters.
2) We propose N-version IR testing, the first systematic approach to finding semantic bugs on binary lifters.
3) We design and implement a fully automated system, called MeanDiff, which can evaluate the correctness of binary lifters. We make our source code public at [3].
4) We test 3 state-of-the-art binary lifters, and demonstrate the effectiveness of our system in terms of finding new semantic bugs.

II. Binary Analysis and IR

This section presents the motivation behind our research by discussing why IR is essential in binary analysis. We first start by defining several terminologies. We then describe the characteristics of IRs, and summarize the current state-of-the-art binary analysis tools.

A. Notation

We let L_{ISA} be an Instruction Set Architecture (ISA), and L_{IR} be an Intermediate Representation (IR). We use a superscript to specify the name of an ISA or an IR. For example, $L_{\text{ISA}}^{\text{x86}}$ means x86 assembly language, and $L_{\text{IR}}^{\text{VEX}}$ is VEX IR. An *instance* of an IR is a sequence of statements defined in the semantics of the IR. We also call each statement in an IR instance as an *IR statement*.

We denote a symbolic execution context by Γ. A symbolic execution context is a map from a variable name to a symbolic expression, e.g., $\Gamma[k]$ denotes the symbolic expression that corresponds to a key k. In our model, we also consider a memory cell as a variable. We denote by $\Gamma' \setminus \Gamma$ an operation over two execution contexts, which returns a map that includes bindings in Γ', but excludes bindings in Γ. A set of keys in Γ can be accessed by `Keys` function: `Keys`(Γ). The number of bindings in an execution context Γ is $|\Gamma|$.

A symbolic evaluation function E_R evaluates an IR statement, generated by a tool R, under a specific execution context. For example, $E_R(s, \Gamma)$ is a function application that evaluates an IR statement $s \in L_{\text{IR}}^R$ under the given execution context Γ, which returns an updated execution context.

B. Representing the Semantics of Binary Code

Understanding binary code is difficult. Although CPU manuals describe the meanings of machine instructions in natural language, there exists no formal specification for them. Furthermore, descriptions vary depending on the version of the manual, and binary instructions typically have implicit semantics that are not obvious from their syntax.

As an example, consider the following x86 instruction that increments the value of the `ECX` register by one: `inc ecx`. Although it is not obvious from the machine code, the instruction can directly affect the value of the `EFLAGS` register. Specifically, there are six status flags in the `EFLAGS` (`ZF`, `CF`, `PF`, `AF`, `SF`, and `OF`), which can change their values based on the computational result of the instruction. Notice, a program can change its control flow depending on the values of the status flags. In addition, the number of affected status flags can also differ depending on the operation. From the example instruction, only five status flags excluding `CF` can change after executing the `inc` instruction.

Binary lifting, denoted by $\uparrow$, is an action that describes the whole semantics of low-level binary code in a high-level Intermediate Representation (IR). We define the term more formally as follows.

Definition 1 (Binary Lifting)**.** Binary lifting is a function $\uparrow_S^R$: $L_{\text{ISA}}^S \rightarrow L_{\text{IR}}^R$, where S is the name of an ISA and R is the name of a target IR.

For example, $\uparrow_{\text{x86}}^{\text{VEX}}$ is a function that takes in x86 binary code as input, and returns a translated VEX instance. $\uparrow_{\text{x86}}^{\text{VEX}}$(`0x41`) returns a lifted IR instance shown in Figure 1b from a binary instruction `0x41`, which is `inc ecx` when decoded. We call a tool that performs binary lifting as a binary lifter. Since the seminal work by Cifuentes *et al.* [19], various binary lifters have been proposed to date.

In this paper, we define a term called *binary-based IR* (BBIR) to distinguish two kinds of IRs: one from binary lifters, and another from compilers. The distinction between BBIRs and classic IRs from compiler theory [8] is mainly from their expressiveness. IRs derived from source code represent high-level language abstractions such as functions and loops. However, BBIRs do not need to consider such language constructs in their abstract syntax tree. Most binary analysis tools such as BAP and Valgrind indeed define their own BBIRs in order to express the low-level semantics of binary code.

TABLE 1
OPEN-SOURCED BINARY-BASED IRS.

IR Name	Tool Name	Explicit	Self-contained	Disasm Dependency	x86	x86-64	FP Support	SIMD Support	Programming Language	Introduction (Year)
BIL	BAP [14]	✓	✓	-	✓	✓	-	◐	OCaml	2011
DBA	BINSEC [11]	✓	✓	-	✓	✗	-	-	OCaml	2011
ESIL	Radare2 [7]	✓	✓	-	✓	✓	-	●	C	2009
LLVM [35]	Remill [55]	✗	✓	-	✓	✓	◐	●	C++	2014
Microcode	Insight [25]	✓	✓	-	✓	✓	-	-	C++	2012
REIL	BinNavi [24]	✓	✓	-	✓	✓	-	-	Java	2008†
Sage III	ROSE [49]	✓	✓	IDA Pro	✓	✓	◐	◐	C++	2007‡
SSL	Boomerang [56]	✓	✓	-	✓	✗	-	-	C++	2004
	Jakstab [32]	✓	✓	-	✓	✗	●	●	Java	2008
TCG	QEMU [12]	✗	✗	-	✓	✓	●	●	C	2009¶
VEX	PyVEX [4]	✗	✗	-	✓	✓	●	●	Python	2013§
	Valgrind [46]	✗	✗	-	✓	✓	●	●	C	2003§
Vine	BitBlaze [53]	✓	✓	-	✓	✓	-	◐	C & OCaml	2008

† The earliest release we found is BinNavi 1.5 in 2008.
‡ The binary support of the ROSE is first presented in 2007.
¶ This is when TCG was first introduced in QEMU [6].
§ The initial version of Valgrind [45] uses an IR called UCode.

```
1  v1 := low:32[ECX]
2  ECX := (low:32[ECX]) + 0x1:32
3  OF := ((high:1[v1]) = (high:1[0x1:32]))
4      & ((high:1[v1]) ^ (high:1[low:32[ECX]]))
5  AF := 0x10:32 = (0x10:32
6                   & ((low:32[ECX] ^ v1) ^ 0x1:32))
7  PF := ~(low:1[let v3 = ((low:32[ECX]) >> 0x4:32)
8                        ^ (low:32[ECX]) in
9                 let v3 = (v3 >> 0x2:32) ^ v3 in
10                (v3 >> 0x1:32) ^ v3])
11 SF := high:1[low:32[ECX]]
12 ZF := 0x0:32 = (low:32[ECX])
```

(a) A lifted IR instance of BAP [14].

```
1  t2 = GET:I32(ecx)
2  t1 = Add32(t2,0x00000001)
3  t3 = GET:I32(cc_op)
4  t4 = GET:I32(cc_dep1)
5  t5 = GET:I32(cc_dep2)
6  t6 = GET:I32(cc_ndep)
7  t7 = x86g_calculate_eflags_c(t3,t4,t5,t6):Ity_I32
8  PUT(cc_ndep) = t7
9  PUT(cc_op) = 0x00000012
10 PUT(cc_dep1) = t1
11 PUT(cc_dep2) = 0x00000000
12 PUT(ecx) = t1
13 PUT(eip) = 0x00000001; Ijk_Boring
```

(b) A lifted IR instance of Valgrind [46].

Fig. 1. BBIR instances lifted from an x86 instruction: `inc ecx`. In (a), `low:32[x]` means taking low 32-bit value from `x`, and hex integers are represented with their bit width.

Definition 2 (Binary-based IR)**.** A BBIR is an IR that is used to represent lifted IR instances from binary code.

C. Current State-of-the-Art Binary Lifters

We survey the existing binary lifters that are open-sourced, and characterize them based on several criteria. As such, we define two notions to describe BBIRs: explicitness and self-containment. Both characteristics play an important role in binary analysis. The explicitness helps in performing control- and data-flow analyses, and the self-containment allows analyzing binaries without having an undesirable over-approximation.

First, the explicitness of a BBIR instance indicates whether each IR statement updates only a single variable in the execution context (Γ). Recall from §II-A that an execution context is a set of variables in our model. Figure 1a shows a BIL instance lifted from an x86 instruction `inc ecx` by BAP. Each statement of the instance can only update a single variable in the execution context. We say, the IR instance is explicit. We now formally define the explicitness of an IR statement as follows.

Definition 3 (IR Explicitness)**.** Given a binary instruction $i \in L^S_{\text{ISA}}$ of an ISA S, an instance of an IR R lifted from i is explicit *iff.* $\forall s \in \uparrow^R_S(i) : |E_R(s, \Gamma) \setminus \Gamma| = 1$.

Being explicit is not always beneficial in terms of expressiveness. For example, VEX has a Compare-And-Swap (CAS) statement, which guarantees an atomic swap operation. Specifically, CAS checks if the target memory has a specific (old) value, and only if the value matches, swap the memory value with a given new value. By definition, compare-and-swap operations, e.g., `xchg` instruction of x86, are not explicit because they change both the source and the destination. Valgrind can easily support such an instruction with a single IR statement because it uses an implicit IR. But, BAP requires multiple IR statements to express such an operation.

Another important criterion we consider is self-containment, which essentially shows whether a lifted IR instance completely explains the semantics of the corresponding binary code. For example, QEMU often relies on external functions to express the semantics of a binary instruction. Consider a logical AND instruction of x86: `pand xmm0, xmm1`. When

we lift the instruction to TCG, the BBIR of QEMU, the IR instance simply passes both register values to an external function called `pand_xmm` instead of articulating its operation within the semantics of the IR. In this case, we say that the IR instance is *not* self-contained, because it has a side-effect.

Definition 4 (IR Self-Containment)**.** Given a binary instruction $i \in L^S_{\text{ISA}}$ of an ISA S, an instance of an IR R is self-contained *iff.* for all IR statement $s \in \uparrow^R_S(i)$, $E_R(s, \Gamma)$ is determined without any side-effect.

Since TCG is designed for CPU emulation, it does not need to be self-contained for its own purpose: external functions in a TCG instance can simply be evaluated at runtime. The same design decision appears in other IRs such as VEX. However, if an IR is not self-contained, analysts need to implement a rule for every external function in the lifter in order to write an analyzer, which requires a significant engineering effort.

Figure 1 shows two BBIR instances lifted from an x86 instruction `inc ecx`. First, the IR instance from BAP (Figure 1a) has seven assignment statements in total including the ones for the status flags of the `EFLAGS`. The IR instance is explicit because every IR statement affects only a single value of the CPU state at a time. The IR instance is self-contained, because the IR instance completely contains the whole semantics of the instruction.

On the other hand, the VEX instance obtained from the same instruction (Figure 1b) using Valgrind is neither explicit nor self-contained. In Line 7, there is a call to an external function `x86g_calculate_eflags_c`, which computes all the status flags and returns the result in a single integer. Since it uses an external function, it is not self-contained. Furthermore, Valgrind does not directly refer to the status flags. Instead, it uses variables (`cc_op`, `cc_dep1`, `cc_dep2`, and `cc_ndep`) that store abstract information about the machine status such as what is the most recently used operation. This is to efficiently compute the `EFLAGS` only when it is needed. However, such an abstract variable represents multiple values (e.g., values of status flags) in the context of a real CPU. Therefore, the IR is implicit.

Table 1 summarizes the current state-of-the-art binary lifters and their BBIRs. The first and the second column of the table show the names of BBIRs and binary lifters respectively. The third column indicates whether a binary lifter emits explicit IR instances. If there is at least one implicit operation in their semantics, we mark them with ✗, and ✓otherwise. The fourth column shows whether a binary lifter produces self-contained IR instances. If a binary lifter can generate an IR instance that has one or more external function calls, we mark it with ✗, and ✓otherwise. When considering explicitness and self-containment of BBIRs, we exclude operations that cannot be modeled without the help of external environments such as system calls. The fifth column shows whether a binary lifter is dependent on IDA pro, a commercial disassembler. There is a lifter (ROSE) that uses the COTS disassembler, but we include it because its IR implementation is open sourced. The sixth and seventh column specify whether a binary lifter can lift x86 and x86-64 instructions respectively. Finally, the eighth and ninth column indicates whether a binary lifter supports floating point and SIMD operations respectively: the ● means a full support, and the ◐ means a partial support.

III. N-VERSION IR TESTING

Given the difficulty of writing the semantics for binary instructions, it is not surprising to see numerous semantic bugs on binary lifters. Even a heavily-tested tool such as QEMU has about 10 bug fixes on their binary lifter every year. This is indeed the primary motivation of our research: we want to build a system that can systematically test the correctness of binary lifters. Most of the binary analysis tools in Table 1 were introduced in the 2000s, and have been used in various areas of research. Therefore, any semantic bugs on binary lifters can have a huge impact on the existing techniques and tools.

We propose a novel testing approach, called *N-version IR testing*, which leverages a symbolic analysis to check the semantic difference between BBIR instances[1]. If one of the IR instances is semantically different from the others, then it means we found a semantic bug in *at least* one of the binary lifters. Once we found a semantic discrepancy, we can manually verify which IR instance is buggy.

To describe our approach, we first define the notion of symbolic equivalence, which is mainly based on that of Person *et al.* [48]. Let S be the name of an ISA, and R be the name of a self-contained BBIR. Suppose a lifted IR instance $\uparrow^R_S(i)$ consists of n statements $\uparrow^R_S(i) = \{s_1, s_2, \ldots, s_n\}$. Then we can evaluate each statement in the IR instance with the evaluation function E_R for an initial execution context Γ to obtain the final execution context Γ':

$$\Gamma' = E_R(s_n, E_R(\ldots, E_R(s_2, E_R(s_1, \Gamma))\ldots)).$$

A symbolic summary of an IR instance shows what the IR instance computes. Intuitively, it can be expressed as a set of updated variable mappings from the execution context after evaluating the IR instance: $\Gamma' \setminus \Gamma$.

Definition 5 (Symbolic Summary (Σ))**.** Given an ISA L^S_{ISA} and an IR L^R_{IR}, a symbolic summary for a binary instruction $i \in L^R_{\text{IR}}$ is

$$\Sigma\left(\uparrow^R_S(i)\right) = \Gamma' \setminus \Gamma$$

where Γ' and Γ are defined as above.

Finally, we say two BBIR instances are semantically equivalent when the output variables of their symbolic summaries have one-to-one correspondence, and each corresponding symbolic summary pair is equivalent to each other.

Definition 6 (Semantic Equivalence of BBIRs)**.** Given two BBIRs R and R', and a binary instruction i of an ISA S, the lifted IR instances $\uparrow^R_S(i)$ and $\uparrow^{R'}_S(i)$ are *semantically equivalent* when:

[1] In this paper, we only test BBIRs generated from a single machine instruction, but the notion of N-version IR testing is general enough to be applied to BBIRs lifted from multiple instructions.

1) $\texttt{Keys}(\Sigma(\uparrow_S^R(i))) = \texttt{Keys}(\Sigma(\uparrow_S^{R'}(i)))$.
2) $\forall k \in \texttt{Keys}(\Sigma(\uparrow_S^R(i))) : \Sigma(\uparrow_S^R(i))[k] \Leftrightarrow \Sigma(\uparrow_S^{R'}(i))[k]$.

The Algorithm. With the above definitions, we present the algorithm of N-version IR testing, a technique to test the correctness of binary lifters. At a high level, N-version IR testing consists of five major steps: StreamGen, Lift, Translate, Summarize, and Triage. Each component behaves as follows.

- **StreamGen:** takes in an ISA L^S_{ISA} as input and returns a sequence of instructions. Since there are too many instructions to consider, we systematically select a set of instructions based on their syntactic structure (§IV-A).
- **Lift:** performs binary lifting $\uparrow_S^R$. Specifically, it takes in an instruction $i \in L^S_{\text{ISA}}$ generated from StreamGen as input, and outputs the corresponding BBIR instance. The target BBIR (R) depends on the lifter we use (§IV-B).
- **Translate:** translates a BBIR instance to a UIR instance, which is a unified intermediate representation that we use in our analysis. The reason we employ this step is mainly due to the different characteristics of BBIRs.
- **Summarize:** takes in a UIR instance as input and returns the corresponding symbolic summary for it. This step includes (1) an input/output variable identification (§IV-C), and (2) a symbolic analysis (§IV-D).
- **Triage:** takes in a set of bugs found, a target instruction, and N distinct symbolic summaries as input. If there is no semantic discrepancy between the symbolic summaries, it simply returns the unmodified set. Otherwise, it returns an updated set that includes the current target instruction, because it is the buggy instruction we found (§IV-E).

The crux of the N-version IR testing algorithm is shown in Algorithm 1. The main function (`testFn`) for N-version IR testing takes in as input a target architecture and a list of binary lifters to test. It outputs a list of semantic bugs found. We note that our algorithm is sound in that we do not report false alarms. However, the algorithm is not complete, because it may miss bugs on the binary lifters under test. For example, when both binary lifters under test have the same buggy implementation, then we may miss the bug.

Example. We now describe the steps in Algorithm 1 with a concrete example. Suppose we are testing two binary lifters on x86: BAP [14] and Valgrind [46]. We let one of the x86 instructions returned from StreamGen in Line 3 is `inc ecx`. Then the algorithm in the for loop (from Line 4 to Line 9 in Algorithm 1) works as follows.

1) We set `summaries` with an empty list (Line 4).
2) Since we assume two binary lifters, the inner for loop (Line 5–9) will iterate two times for each binary lifter.
3) In Line 6, we lift the binary instruction to a BIL instance (Figure 1a). We then translate the IR instance to a UIR instance (Line 7). If the source IR is implicit or not self-contained, the translation becomes challenging. We address this issue in §IV-B.
4) We then compute the symbolic summary, which maps output variables to symbolic expressions, from the translated IR instance, which consists of two major steps: a) variable identification, and b) symbolic execution.
   a) From the IR instance, we identify the `ECX` register as both an input and an output variable, and 5 status flags (`OF`, `AF`, `PF`, `SF`, and `ZF`) as output variables. To achieve this, we perform a simple data-flow analysis: see §IV-C.
   b) Once we identify the output variables, we run symbolic execution on the IR instance to obtain symbolic summaries for each of the output variables. In this case, the only input variable is `ECX`, so we let the variable as symbolic, and evaluate the IR instance. For example, in Line 2 of Figure 1a, the output variable `ECX` has a symbolic expression `ECX + 1`. In this way, each output variable has the corresponding symbolic expression of the input variable(s), e.g., $\texttt{ECX} \mapsto \texttt{ECX + 1}$.
5) In Line 9, we add the obtained symbolic summary to `summaries`, and repeat this process for each lifter.
6) In Line 10, we check the semantic equivalence between symbolic summaries obtained from the previous steps. For simplicity, let us assume that each BBIR instance has only a single output variable `ECX`, i.e., we do not consider the `EFLAGS` register. Then, the final symbolic summaries from both BAP and Valgrind for `ECX` would be `ECX + 1`. Recall from Definition 6, we consider two conditions to decide whether the symbolic summaries are semantically equivalent. First, we check whether the set of output variable names are equivalent. Second, we check for each output variable whether the corresponding symbolic summaries are equivalent: $\texttt{ECX + 1} \Leftrightarrow \texttt{ECX + 1}$. In this example, we conclude that both BAP and Valgrind emits semantically equivalent IR instances from the given instruction.

Algorithm 1: N-version IR testing

```
1  function testFn(L^S_ISA, lifters)
2      bugs ← ∅
3      for i in StreamGen (L^S_ISA) do
4          summaries ← [·] // An empty list
5          for ↑^R_S in lifters do
6              bbir = ↑^R_S (i) // Lift
7              uir = Translate (bbir)
8              s = Σ(uir) // Summarize
9              summaries ← summaries + s
10         bugs ← Triage (bugs, i, summaries)
11     return bugs
```

IV. Design

In this section, we describe the design of MeanDiff, an automated system that implements the N-version IR testing. Following the focus of MeanDiff at a higher level, we provide a detailed review of the challenges this research project has overcome. Using the pipeline, illustrated in Figure 2, these challenges will be described in a chronological order, following the process from input to output.

A. Instruction Stream Generation

MeanDiff checks the semantic equivalence per each instruction returned from STREAMGEN. Ideally, one can generate every possible instruction of a given architecture in order to completely test BBIRs. However, this is infeasible due to the huge number of possible instructions to consider. For example, there are more than 2^{32} `add` instructions on x86 even though we only consider the ones that have `EAX` as the destination operand: "`add eax, 0x0`", "`add eax, 0x1`", ..., "`add eax, 0xffffffff`", "`add eax, eax`", "`add eax, ebx`", and so forth.

Notice, this naïve approach already requires radically few test cases compared to existing differential testing approaches [43], [44], because N-version IR testing does *not* require employing test cases for all possible states of each instruction. Particularly, a symbolic summary for a given IR instance encapsulates the semantics of the instance for all possible input values.

However, we can further reduce the number of test cases to consider by exploiting the nature of symbolic evaluation. Specifically, instructions with the same opcode, but with different register names will end up having the symbolic summaries that are syntactically similar: only the name of the symbols are different. For example, both `add eax, ebx` and `add ecx, edx` will produce two symbolic summaries that produce the same result when applied to N-version IR testing. With this intuition, MeanDiff generates test cases for every combination of available operand types of a given opcode as follows.

1) For operand type `reg, reg`, we generate two test cases: one with the same register, another with different registers for each operand.
2) For operand type `reg, mem` and `mem, reg`, we generate single test case for every possible addressing mode. That is, we consider `[reg]`, `[reg + reg]`, `[reg + displacement]`, `[reg + reg + displacement]`, and so forth. We use an arbitrary value for the displacement.
3) For operand type `reg, imm` and `mem, imm`, we pick three constant values for the immediate operand to generate test cases: 0, 42, and the maximum unsigned value based on the bit width of the immediate. For example, we consider the following three cases: `add al, 0x0`, `add al, 0x42`, and `add al, 0xff`. This is to cover semantic errors that are triggered only when the immediate has a specific value.

While generating test instructions, STREAMGEN removes redundant instructions as different opcodes may be decoded to the same instruction on x86. For example, `0x0118` and `0x011c20` are both `add [eax], ebx` on x86. In our STREAMGEN implementation, it generates 323,928 and 1,161,430 valid instructions on x86 and x86-64 respectively.

B. IR Lifting and Translation

Obtaining BBIR instances from binary lifters requires varying amounts of manual effort. Some systems such as BAP [14] and BINSEC [11] provide APIs to lift binary code to their own BBIR instance as well as to access the Abstract Syntax Tree (AST) nodes of BBIR instances. However, other systems such as Valgrind [46] do not provide such functionalities. PyVEX [4] ships with the manually extracted VEX module from the Valgrind project, which provides a python API to access the ASTs. Our LIFT implementation uses PyVEX.

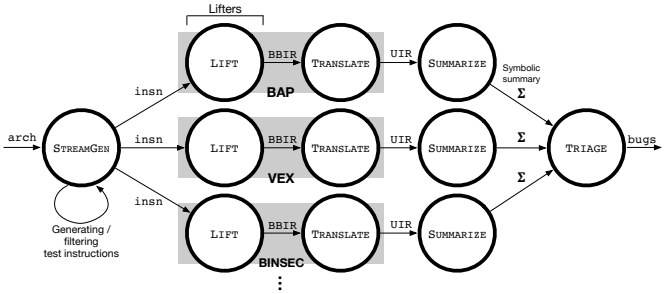

Fig. 2. MeanDiff Architecture.

As discussed in §II-C, the spectrum of semantics supported by the BBIRs differs. Specifically, we consider the explicitness and the self-containment of BBIRs. First, the explicitness of each IR varies. In BBIRs such as BIL and DBA, the `EFLAGS` register is represented by explicitly declaring each flag as an output variable. VEX from PyVEX, on the other hand, uses a condition code register which is used through lazy evaluation. To handle this, the `EFLAGS` has to be manually computed for each instruction. This makes the semantics conform with that of BAP and BINSEC. Similarly, there are some implicit operations such as compare and swap (CAS) that are only supported by some BBIRs.

There is also the question whether the BBIR at hand is self-contained, following Definition 4 in §II-C. An IR that is not self-contained may call an external function within the IR instance. Recall from the example in Figure 1b, the external function call to `x86g_calculate_eflags_c` within the IR instance makes it difficult for us to compare the semantics of it with other IR instances.

To handle these challenges, we define a Unified Intermediate Representation, UIR, used for unifying every BBIR into a single form. UIR is a simple, but Turing-complete language, which consists of a few primitive arithmetic and logical operations. It is also designed to be explicit and self-contained. Due to the page limit, we do not show the detailed syntax of UIR. Interested readers should refer to our web page [3]. We note that employing a unified representation benefits the rest of our analysis as well: we do not need to write the same analysis routine for every BBIR under test.

Translating BBIR to UIR is challenging and requires significant engineering efforts. For example, we replaced the external functions of VEX that compute the `EFLAGS`. In our experiment, we employed a conservative approach instead of fully implementing the conversion from BBIRs to UIR. Particularly, whenever we encounter a BBIR instance that is not explicit or not self-contained during the analysis, we omit

such an instruction from our testing set unless it is specifically handled by our translators (TRANSLATE). We leave it as future work to shrink the semantic gaps between different BBIRs by implementing more conversion rules. We note, however, that omitting such instructions do not affect the soundness guarantee of our analysis.

C. Data Flow Analysis

The first step of SUMMARIZE is to identify input/output variables from a translated UIR instance. To determine input variables from an IR instance, we apply a classic use-def analysis. Every node in the Use-Def (UD) chain that has outgoing edges, but no incoming edges, is classified as an input variable. In other words, variables that are used, but never defined are input variables.

On the other hand, we cannot simply identify output variables with a data-flow analysis. One may think that all variables that are declared, but never read, must be output variables. However, this is not the case for many IRs. For example, in Figure 1, `ECX` is an output variable while also being used in the assignment of `OF`. To overcome this challenge, we use the knowledge of the given ISA. Specifically, we check for every statement s in the given IR instance whether s assigns a value to a variable that has the name of known registers and status flags. If so, we make all such variables to be an output.

D. Symbolic Execution

Recall from §III, the second step of SUMMARIZE is to symbolically execute each of the lifted IR instances in order to generate symbolic summaries. We address two challenges in this phase: (1) loop handling challenge, and (2) symbolic memory challenge. The first challenge is simply handled by unrolling loops once. Although we can only see a limited number of execution paths, our analysis is still sound: it does not produce false alarms. In this subsection, we focus on how we handle the second challenge.

Symbolic memory is a traditional problem in symbolic execution where a symbolic address is used to access a memory value. One may represent a symbolic memory access with an if-then-else chain accounting for every possible values in the memory, but this may result in too complex symbolic expressions to be solved in practice. There are several existing symbolic access policies [50] in dynamic symbolic execution such as pointer concretization, but this is not an option for our approach since we rely on static symbolic execution.

In our implementation, however, the symbolic memory challenge is not really an issue, because MeanDiff only focuses on BBIRs generated from a single machine instruction (Recall §III). Notice, it is extremely unlikely to have a machine instruction that uses a non-memory operand to access memory. Furthermore, the address of a memory operand is typically not used for other purposes than for accessing the memory.

Therefore, we can simply let a memory access be a symbolic variable if it is used as input. When lifting an x86 instruction, "`mov ebx, [eax]`", for example, MeanDiff replaces the `[eax]` operand with a symbolic variable `mem_EAX`. However, simply replacing memory expressions into a symbolic variable can be problematic when two different lifters express the same memory operand in a totally different manner. For example, the address of a memory operand `[eax + ebx]` can be represented either as `eax + ebx` or as `ebx + eax`. Furthermore, a single instruction may access more than one memory cells. This means we need to be able to distinguish memory expressions, and give a unique symbolic name for the same memory expressions.

To determine if two memory expressions are the same, we apply a series of simple transformation rules such as expression reordering ($e1 + e2 = e2 + e1$), strength reduction ($e \times 2^n = e << n$) and arithmetic simplification ($e + 0 = e$) to both, and check if the expressions are exactly the same.

E. Triage

The final step of N-version IR testing is TRIAGE, which checks the semantic equivalence between symbolic summaries obtained from SUMMARIZE. Given N symbolic summaries, MeanDiff first checks if they have the same set of output variables. If not, we add the corresponding instruction to our bug database. Otherwise, MeanDiff constructs a final formula by combining symbolic expressions for each of the output variables. It solves the formula using the Z3 SMT solver [23]. If we found any counter-example that gives two or more distinct values when evaluating symbolic expressions of the same output variable, it means we found a bug.

The Intel manual specifies that register values can be marked as "`undefined`" after executing an instruction. For example, the `AF` flag is undefined after executing logical operations. Some lifters such as BAP explicitly marks the `AF` as undefined in this case, but others such as BINSEC would simply let the `AF` unchanged. In order to handle such a case, we do not compare undefined output variables in our implementation.

F. Implementation

We applied N-version IR testing on 3 existing binary analysis tools in Table 1: BAP, PyVEX (a wrapper for Valgrind's VEX), and BINSEC. We have implemented translators (TRANSLATE) for each of the tools: 400 lines of OCaml for a BIL-to-UIR translator; 300 lines of OCaml for a DBA-to-UIR translator, and 2.0k lines of Python for a VEX-to-UIR translator. We have also implemented our SUMMARIZE and TRIAGE module with about 2.0K lines of F# code. We made the source code public at [3].

V. EVALUATION

To evaluate MeanDiff, we focus on the following questions:

1) How many binary instructions can STREAMGEN generate? And how many of them can be handled by the-state-of-the-art binary lifters?
2) Can MeanDiff find semantic bugs in the binary lifters?
3) How do the bugs look like? And how difficult it is to write precise binary lifters?

A. Environment Setup

We ran our experiments on 64-bit Ubuntu 16.04.3 system, with 2 22-core CPUs, Intel Xeon E5-1600 family, with 256GB RAM. We downloaded three binary analysis tools, i.e., binary lifters: (1) BAP 1.2.0 (released Feb. 10th, 2017); (2) PyVEX 6.7.4.12 (released Apr. 12th, 2017); and (3) BINSEC 0.1 (released Mar. 1st, 2017). All the numbers reported in this paper is based on the experimental results on these binary lifters. We tested the lifters on both x86 and x86-64 instructions, with the exception of BINSEC that does not support an x86 ISA. The total experiments took approximately 2 days.

B. Binary Lifting

Recall from §IV-A, our StreamGen generated 323,928 and 1,161,430 instructions on x86 and x86-64 respectively. Firstly, we recorded the numbers of successfully lifted BBIRs for each binary lifter under test to later use them in the evaluation of MeanDiff.

Table 2 shows the result. Each row represents the number of lifted instructions that could only be successfully lifted from the specified set of lifters. From these results, it is apparent that BINSEC could lift the smallest number of instructions. There is also a noticeable clustering of instructions which only BAP and BINSEC could lift. This is due to the instruction prefix, `f3 : rep`, which is used for expressing the instruction level loop. From the Intel Developer's Manual [2], if the `rep` prefix is used with an inappropriate opcode, the prefix should be ignored. However, PyVEX simply refuses to interpret such instructions. That is why such a huge number of instructions was lifted with BAP and BINSEC in comparison with PyVEX.

The "None" row of the table indicates the number of instructions that are *not* successfully lifted from any one of the binary lifters under test. Almost 16% and 33% of the instructions on x86 and x86-64, respectively, could not be lifted by the binary lifters. This result signifies two points: (1) only a small subset of available instructions are used in binary executable in practice; and (2) the current state-of-the-art lifters are yet to be perfect.

C. Bugs Found

Can MeanDiff find realistic semantic bugs? To answer the question, we checked the semantic equivalence on every lifted BBIRs under test.

In total, MeanDiff found semantic bugs from 65,940 distinct binary instructions. Since two or more instructions can cause the same semantic bug, we manually verified all the bugs we found, and obtained 24 unique bugs in total. More specifically, we found 23 unique bugs on x86, and 10 unique bugs on x86-64. Out of 10 unique x86-64 bugs, 9 bugs were overlapping with x86 bugs. Thus, we did not count them. Table 3 shows the list of bugs that we found with MeanDiff. The example column shows a sample instruction that you can trigger the bug. In the table, there are some cases that the same type of bug occurs in different lifters. We treat them as a separate bug since the implementations of lifters are different.

TABLE 2
The number of successfully lifted binary instructions.

Lifter(s)	x86	x86-64
PyVEX & BAP & BINSEC	71,350	-
PyVEX & BAP	85,066	286,081
PyVEX & BINSEC	81,872	-
BAP & BINSEC	175,416	-
PyVEX	135,172	516,974
BAP	206,118	632,035
BINSEC	202,652	-
None	50,990	358,375

D. Case Studies

In this subsection, we examine three interesting cases of semantic bugs found by MeanDiff.

Case Study #1 (`push`). Stack operations such as `push` and `pop` are used in various execution contexts: local and temporary variables for functions are stored on the stack; function arguments are passed through the stack; and return addresses are saved on the stack. Given that stack operations are so common, the semantics of the `push` instruction should be correct?

The Intel Developer's Manual [2] describes the semantics of `push`, but most of the description is written in natural language, while only the operation segment is expressed in pseudo code. Although the manual does its best to describe every possible semantic, it is often not possible to understand all possible corner cases by just looking at the pseudo code.

The binary sequence `6aff` is translated into `push 0xff`. Since the stack pointer never decreases by 1, the size of source operand is smaller than the size of the operation. If one tries to look up how the CPU deals with this situation, one may fail to locate the pseudo-code. Instead, it is indicated in the description written in English. The manual indicates in this situation that the source operand must be sign-extended before it is pushed to the stack.

Figure 3 describes the result of each target binary lifter, which shows how each lifter handles the situation described above. Both BAP and BINSEC produce BBIR instances with semantics that move the 32-bit value `0xff` into memory. PyVEX, on the other hand, produces an instance, which moves the 32-bit value `0xffffffff` into memory. This is an obvious semantic discrepancy, which symbolizes the difficulty to implement binary-based tools that have the precise meaning.

Case Study #2 (`bt`). MeanDiff is even able to find bugs in PyVEX, which has been extensively tested and applied in the state-of-the-art research The instruction `bt`, which stands for Bit Test, has two operands. It tests the bit in the first operand, at the index specified in the second operand, and puts the result in the CF status flag. The target binary instruction is `0fa3d8`, which is translated to `bt eax, ebx`. Unfortunately, MeanDiff is not able to test all three binary lifters as BINSEC does not understand this instruction. Thus, it only compares the BBIR instances from BAP and PyVEX.

TABLE 3
BUGS FOUND

Lifter	Bug Description	Example	Opcode
BAP	Confusion of operands	0fc100	xadd
BAP	Operand address changed	0fc100	xadd
BAP	Not taking mod size	c000ff	rol
BAP	Missing signed extension	6aff	push
BAP	Invalid CF calculation	0fc1c0	xadd
BAP	Invalid OF calculation	28c0	sub
PyVEX	Missing arithmetic operation	0fc1c0	xadd
PyVEX	Useless memory access	0fa3c0	bt
PyVEX	Invalid push/pop operation	0e	push
PyVEX	Not storing segment register	488c00	mov
BINSEC	Confusion of operands	0f3a0f0042	palingr
BINSEC	Operand address changed	0fc000	xadd
BINSEC	Missing signed extension	6b08ff	add
BINSEC	Invalid padding	678cc0	mov
BINSEC	Invalid order of calculation	0fb100	cmpxchg
BINSEC	Invalid memory addr. calculation	0fa300	bt
BINSEC	Not taking part of value	0fa300	bt
BINSEC	Unable to recognize prefix	64668d08	lea
BINSEC	Invalid pop operation	5c	pop
BINSEC	Invalid pushad behavior	60	pushad
BINSEC	Invalid store/load AF behavior	9e	sahf
BINSEC	Invalid CF calculation	1cff	sbb
BINSEC	Invalid OF calculation	1cff	sbb
BINSEC	Operand address changed	0fc000	xadd

```
v1 := 0x80:32
ESP := ESP - 0x4:32
mem32 := mem32 with [ESP, el]:u32 <- v1
```

(a) Lifted IR from BAP [14].

```
t3 = GET:I32(esp)
t2 = Sub32(t3,0x00000004)
PUT(esp) = t2
STle(t2) = 0xffffff80
```

(b) Lifted IR from PyVEX [4].

```
@[(esp(32) - 4(32))]L4 := 255(32)
esp := (esp(32) - 4(32))
```

(c) Lifted IR from BINSEC [11].

Fig. 3. Bug case study #1: `push 0xff`.

```
o1 := low:5[EBX]
CF := low:1[EAX >> o1]
OF := unknown[bt]:u1
SF := unknown[bt]:u1
ZF := unknown[bt]:u1
AF := unknown[bt]:u1
PF := unknown[bt]:u1
```

(a) Lifted IR from BAP [14].

```
t2 = GET:I32(ebx)
t9 = GET:I32(esp)
t8 = Sub32(t9,0x00000080)
PUT(esp) = t8
t10 = GET:I32(eax)
STle(t8) = t10
t3 = And32(t2,0x0000001f)
t12 = Sar32(t3,0x03)
t11 = Add32(t8,t12)
t14 = And32(t3,0x00000007)
t13 = 32to8(t14)
t0 = LDle:I8(t11)
PUT(cc_op) = 0x00000000
PUT(cc_dep2) = 0x00000000
t17 = 8Uto32(t0)
t16 = Shr32(t17,t13)
t15 = And32(t16,
        0x00000001)
PUT(cc_dep1) = t15
PUT(cc_ndep) = 0x00000000
t18 = LDle:I32(t8)
PUT(eax) = t18
t19 = Add32(t8,0x00000080)
PUT(esp) = t19
```

(b) Lifted IR from PyVEX [4].

Fig. 4. Bug case study #2: `bt eax, ebx`.

Figure 4 shows the lifted BBIR instances. By comparing the difference in the output size, one may begin to realize the difference in semantics. BAP clearly bit-shifts EAX to right by the number specified in EBX, followed by storing the lowest bit in CF. This is indeed the semantics defined in Intel Developer's Manual [2]

However, the VEX IR instance starts by decrementing the stack pointer by 0x80 and storing the value of EAX at the resulting address. Further, it computes the actual bit test, which is stored in cc_dep1. Lastly it restores the stack pointer by incrementing, but without restoring the value it previously modified at −0x80(ESP). In other words, the CPU state has been altered in a way not defined in the x86 instruction definition. The discrepancy may seem obvious when directly comparing it to another BBIR, but due to the sheer amount of instructions in modern architectures, bugs of this nature are difficult to detect.

Case Study #3 (xadd). The last case introduced in this paper is the xadd; exchange and add. This case is interesting as both BAP and BINSEC confuse the semantics regarding the operands in a similar, but different fashion.

First, let us introduce the binary instruction: 0fc100. It is a combination of the opcode 0fc1 and operand 00, which represents EAX as source and [EAX] as destination operand. The semantics of this instruction listed in the manual [2] is as follows: exchange the value of destination and source operand, and then load the sum of the two values into the destination operand. Now, let us see how each binary lifter represents this semantics in their own BBIR.

Figure 5 shows that each of the lifted BBIR instances represents different semantics. Starting with the last, i.e. the DBA instance in Figure 5c, it calculates the correct value and corresponding flags, but somehow changes the destination address before writing the result to it. The BIL instance, in Figure 5a, correctly calculates the result, but switches the operands such that the result is written to the source operand.

Interestingly enough, there is another bug present in the core of calculating status flags. For example, OF is calculated based on the memory operand, which has already been changed. The only correct instance is from PyVEX, in Figure 5b, which correctly writes the result to the correct operand.

The Lesson. From the above examples, we have shown that even the heavily tested binary lifters have semantic bugs. It is extremely difficult to consider every possible semantic details

```
v1 := (low:32[EAX])
        + (mem32[pad:32[low:32[EAX]], el]:u32)
mem32 := mem32 with [pad:32[low:32[EAX]], el]:u32
           <- low:32[EAX]
EAX := v1
...
OF := ((high:1[v1]) =
        (high:1[mem32[pad:32[low:32[EAX]], el]:u32]))
        & ((high:1[v1]) ^ (high:1[low:32[EAX]]))
AF := 0x10:32 = (0x10:32 \& (((low:32[EAX]) ^ v1)
        ^ (mem32[pad:32[low:32[EAX]], el]:u32)))
...
```

(a) Lifted IR from BAP [14].

```
t3 = GET:I32(eax)
t0 = LDle:I32(t3)
t2 = Add32(t0,t3)
STle(t3) = t2
PUT(cc_op) = 0x00000003
PUT(cc_dep1) = t0
PUT(cc_dep2) = t3
PUT(cc_ndep) =
  0x00000000
PUT(eax) = t0
```

(b) Lifted IR from PyVEX [4].

```
res32 := (@[eax(32)] + eax(32))
OF := ((@[eax(32)]L4{31} = eax(32){31}) &
    (@[eax(32)]L4{31} != res32(32){31}))
...
eax := @[eax(32)]
@[eax(32)] := res32(32)
```

(c) Lifted IR from BINSEC [11].

Fig. 5. Bug case study #3: `xadd [eax],eax`.

by simply reading the manual.

VI. Discussion and Limitation

In this paper, we tested three of the binary lifters listed in Table 1. Recall from §IV-B, extracting a binary lifter from existing binary analysis tools can require significant manual effort. We handle this issue by making MeanDiff as an open source project. Binary analysts who have their own lifter, or borrow a famous lifter from another project, will be able to add their lifter to MeanDiff and test the correctness of it.

MeanDiff currently does not test instructions performing floating point operations. As shown in Table 1, many lifters, including BAP and BINSEC, do not support floating point operations. As such, we currently do not take floating point operations into account. However, it is straightforward to modify UIR to handle floating point operations by using a theory of floating-point numbers in SMT solving [51]. We leave it as future work to support such functionality.

MeanDiff currently supports only x86 and x86-64. To add support for another ISA, one needs to build a new stream generator for the ISA by manually analyzing the syntactic structure of every instruction in the ISA. Indeed, one of the reasons why we make our source code public is to encourage the community to adapt this technique to test the correctness of various BBIRs.

Our current focus is on BBIR instances generated from a single instruction, but we can potentially extend MeanDiff to handle BBIR instances from multiple instructions. Since some lifters such as PyVEX performs intra basic-block optimizations, we may be able to find interesting semantic bugs by extending our scope. However, by considering BBIR instances from multiple instructions, we may face several challenges. First, StreamGen needs to consider all possible combinations of instructions. Second, the classic symbolic memory challenge may occur frequently (recall from §IV-D). We believe this is a promising direction for future research.

VII. Related Work

The idea of a symbolic equivalence check itself is not new [22], [34], [41], [48], but our work is the first attempt to applying the idea to testing the correctness of BBIRs. Luo *et al.* [41] recently extended the idea of a symbolic equivalence check to perform similarity comparison on obfuscated binary code. We believe that N-version IR testing can contribute to solving the addressed challenges by providing more accurate semantics of binary code.

Hasabnis *et al.* [29] attempt to test IRs generated by compilers. The key difference between their work and ours is that they rely on the CPU to test the correctness. They compare the result from CPU with the result from an IR emulator. There are several pieces of work in this line of research: [43], [44]. All the existing techniques rely on the actual CPU state and randomly generated test cases.

One remarkable attempt, in the area of binary lifting, is automatically generating BBIRs. Godefroid [28] *et al.* made this problem into the program synthesis problem with a black-box oracle. They divided ALU operations into 3 groups, made templates for each group, and synthesized IRs from those templates. The automatic BBIR generation problem was turned into a problem in the area of machine learning by Hasabnis [30] *et al.*. They used the problem of learning a parameterized translation on trees to automatically synthesize BBIRs. These approaches do not guarantee the semantic correctness of the generated BBIRs. Thus, our approach is orthogonal and complementary to their techniques.

VIII. Conclusion

In this paper we proposed N-version IR testing, a novel technique to find semantic bugs in binary lifters. We systematically studied existing binary lifters to motivate our research, and addressed several challenges in applying our technique to binary-based IRs. We implemented the proposed technique in MeanDiff, and evaluated the system on 3 state-of-the-art binary lifters. We found 24 unique semantic bugs, which were all manually confirmed. Furthermore, we have reported all of our findings to the developers of the tested binary lifters. Our results indicate that any binary analysis can go wrong even with a well-founded theory when the semantics of binary-based IRs is wrong.

Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.B0717-16-0109, Building a Platform for Automated Reverse Engineering and Vulnerability Detection with Binary Code Analysis).

REFERENCES

[1] "ARM architecture reference manual," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html.

[2] "Intel® 64 and ia-32 architectures software developer's manual," https://software.intel.com/en-us/articles/intel-sdm.

[3] "MeanDiff," https://github.com/SoftSec-KAIST/MeanDiff.

[4] "Pyvex," https://github.com/angr/pyvex.

[5] "QEMU-devel archives," http://lists.gnu.org/archive/html/qemu-devel/2017-01/msg03062.html.

[6] "QEMU-devel archives," http://lists.gnu.org/archive/html/qemu-devel/2009-03/msg00154.html.

[7] "Radare2," https://github.com/radare/radare2.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison Wesley.

[9] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with Veritesting," in *Proceedings of the International Conference on Software Engineering*, 2014, pp. 1083–1094.

[10] G. Balakrishnan and T. Reps, "WYSINWYX: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 6, pp. 23:1–23:84, 2010.

[11] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The BINCOA framework for binary code analysis," in *Proceedings of the International Conference on Computer Aided Verification*, 2011, pp. 165–170.

[12] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference*, 2005, pp. 41–46.

[13] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2003, pp. 265–275.

[14] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proceedings of the International Conference on Computer Aided Verification*, 2011, pp. 463–469.

[15] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2012, pp. 380–394.

[16] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2015, pp. 725–741.

[17] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 265–278.

[18] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2005, pp. 32–46.

[19] C. Cifuentes and S. Sendall, "Specifying the semantics of machine instructions," in *Proceedings of the International Workshop on Program Comprehension*, 1998, pp. 126–133.

[20] C. Cifuentes and M. V. Emmerik, "UQBT: Adaptable binary translation at low cost," *Computer*, vol. 33, no. 3, pp. 60–66, 2000.

[21] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2008, pp. 391–402.

[22] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan, "Embedded software verification using symbolic execution and uninterpreted functions," *International Journal of Parallel Programming*, vol. 34, no. 1, pp. 61–91, 2006.

[23] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[24] T. Dullien and S. Porst, "REIL: A platform-independent intermediate representation of disassembled code for static code analysis," in *Proceedings of CanSecWest*, 2009.

[25] E. Fleury, O. Ly, G. Point, and A. Vincent, "Insight: An open binary analysis framework," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2015, pp. 218–224.

[26] B. Ford and R. Cox, "Vx32: Lightweight user-level sandboxing on the x86," in *Proceedings of the USENIX Annual Technical Conference*, 2008, pp. 293–306.

[27] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[28] P. Godefroid and A. Taly, "Automated synthesis of symbolic instruction encodings from i/o samples," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2012, pp. 441–452.

[29] N. Hasabnis, R. Qiao, and R. Sekar, "Checking correctness of code generator architecture specifications," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2015, pp. 167–178.

[30] N. Hasabnis and R. Sekar, "Lifting assembly to intermediate representation: A novel approach leveraging compilers," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 311–324.

[31] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 2, pp. 123–193, 1999.

[32] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Proceedings of the International Conference on Computer Aided Verification*, 2008, pp. 423–427.

[33] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis," in *Proceedings of the Annual Computer Security Applications Conference*, 2004, pp. 91–100.

[34] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "SYMDIFF: A language-agnostic semantic diff tool for imperative programs," in *Proceedings of the International Conference on Computer Aided Verification*, 2012, pp. 712–717.

[35] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, pp. 75–87.

[36] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for linux," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software*, 2010, pp. 175–183.

[37] K. P. Lawton, "Bochs: A portable PC emulator for Unix/X," *Linux Journal*, vol. 1996, no. 29es, 1996.

[38] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in *Proceedings of the Network and Distributed System Security Symposium*, 2011, pp. 251–268.

[39] Z. Lin and X. Zhang, "Deriving input syntactic structure from execution," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2008, pp. 83–93.

[40] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2005, pp. 190–200.

[41] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2014, pp. 389–400.

[42] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[43] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi, "Testing system virtual machines," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2010, pp. 171–182.

[44] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing cpu emulators," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2009, pp. 261–272.

[45] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," in *Proceedings of the Workshop on Runtime Verification*, 2003.

[46] ——, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2007, pp. 89–100.

[47] R. Paleari, L. Martignoni, G. Fresi Roglia, and D. Bruschi, "N-version disassembly: Differential testing of x86 disassemblers," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2010, pp. 265–274.

[48] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2008, pp. 226–237.

[49] D. Quinlan, G. Barany, and T. Panas, "Shared and distributed memory parallel security analysis of large-scale source code and binary applications," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep., 2007.

[50] A. Romano and D. R. Engler, "symMMU: Symbolically executed runtime libraries for symbolic memory access," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2014, pp. 247–258.

[51] P. Rümmer and T. Wahl, "An SMT-LIB theory of binary floating-point arithmetic," in *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland*, 2010.

[52] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Proceedings of the USENIX Security Symposium*, 2013, pp. 353–368.

[53] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Proceedings of the International Conference on Information Systems Security*, 2008.

[54] K. Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984.

[55] Trail of Bits, "Remill," https://github.com/trailofbits/remill.

[56] M. Van Emmerik and T. Waddington, "Using a decompiler for real-world source recovery," in *Proceedings of the Working Conference on Reverse Engineering*, 2004, pp. 27–36.

[57] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations," in *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[58] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2007, pp. 116–127.

[59] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2013, pp. 559–573.

[60] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proceedings of the USENIX Security Symposium*, 2013, pp. 337–352.