LiftFuzz: Validating Binary Lifters through Context-aware Fuzzing with GPT

Yutong Zhou
The Chinese University of Hong Kong
Hong Kong SAR, China
zy319@ie.cuhk.edu.hk

Fan Yang
The Chinese University of Hong Kong
Hong Kong SAR, China
yf020@ie.cuhk.edu.hk

Zirui Song
The Chinese University of Hong Kong
Hong Kong SAR, China
sz019@ie.cuhk.edu.hk

Ke Zhang
The Chinese University of Hong Kong
Hong Kong SAR, China
zk019@ie.cuhk.edu.hk

Jiongyi Chen
National University of Defense Technology
Changsha, China
chenjiongyi@nudt.edu.cn

Kehuan Zhang
The Chinese University of Hong Kong
Hong Kong SAR, China
khzhang@ie.cuhk.edu.hk

ABSTRACT

Analyzing binary code is vital for software engineering and security research, particularly when the source code is unavailable. However, understanding, modifying, and retargeting binary code can be complex tasks. To counter these difficulties, binary lifters have been introduced. These tools translate binary code into Intermediate Representations (IRs), providing several advantages, such as enabling modifications to executables without source code and facilitating code retargetability. So far, accurately developing binary lifters for modern ISAs is universally acknowledged as challenging and error-prone. Existing validation methods mainly concentrate on isolated instructions, overlooking interactions among instructions. In this paper, we introduce LiftFuzz, a novel framework that leverages instruction context-aware fuzzing to validate binary lifters. LiftFuzz harnesses an assembly language model to learn interactions among instructions and generates test cases with the knowledge. LiftFuzz greatly outperforms the baseline, requiring only 1/1000 of the test cases used by the baseline to identify 26 inconsistencies, including a previously uncovered category. LiftFuzz significantly contributes to enhancing the performance of binary lifters, which are frequently employed in binary security applications.

CCS CONCEPTS

• **Security and privacy → Software and application security**; **Software reverse engineering**;

KEYWORDS

Binary Code Analysis, Binary Lifter, Fuzzing, Machine Learning

ACM Reference Format:
Yutong Zhou, Fan Yang, Zirui Song, Ke Zhang, Jiongyi Chen, and Kehuan Zhang. 2024. LiftFuzz: Validating Binary Lifters through Context-aware Fuzzing with GPT . In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670276

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10.
https://doi.org/10.1145/3658644.3670276

1 INTRODUCTION

Binary analysis of executable codes is of utmost importance in software engineering and security research [6, 7, 11, 19, 28, 34]. This significance arises not only from real-world scenarios where source code may not be accessible, such as in IoT firmware, malware, legacy code, or Commercial-Off-The-Shelf (COTS) software, but also since compilers themselves are not exempt from bugs or errors [49, 55] and toolchains can be contaminated, *e.g.*, Xcode Ghost [54].

The initial step in binary analysis involves converting a binary executable into a sequence of instructions through the process of disassembly. However, the output of disassemblers is still challenging for humans to comprehend and analyze effectively. Besides, the disassembly output is specific to the architecture and cannot be easily retargeted to different platforms. For further reverse engineering, the binary lifter is proposed to translate the assembly instructions into an intermediate representation (IR), which enables the users to comprehend, patch, and modify the binary executable without accessing the source code.

Naturally, the IR obtained from the binary lifter serves as the foundation for downstream binary analysis tools. Hence, binary lifters are frequently employed for tasks that demand the utmost precision, such as identifying security vulnerabilities in the binary executable. Consequently, any bug or error in the converted IR can lead to the failure of subsequent binary analysis processes. Regrettably, developing an accurate binary lifter to support intricate modern Instruction Set Architectures (ISAs) [23] is challenging and error-prone. This is primarily due to the vast amount of information contained within an instruction set manual, making it difficult to fully comprehend and encode the effects of numerous instructions. Compounding the challenge is the constant need for maintenance, as new instructions are continually introduced to support evolving CPU features. Furthermore, the specifications provided by hardware manufacturers are often unreliable and may contain mistakes or lack comprehensive informational semantics for CPU instructions [13]. Therefore, it is non-trivial to validate the correctness of binary lifters.

Unfortunately, the aforementioned challenges in implementing the binary lifter also make the validation challenging. For instance,

the massive instructions within an ISA result in a wide range of testing. Although there are many efforts such as Kim et al. [27] and Dasgupta et al. [12] are proposed to validate the binary lifter, all of them only focused on isolated instruction validation, which means none of them have considered the interactions among instructions. Based on our observation, the binary lifter produces different IRs for the same instruction due to varying contexts. While some IRs are faithful, others may not, suggesting that validating each instruction in isolation is insufficient. Therefore, it is crucial to assess the binary lifter's capacity to analyze the instruction context and manage the interactions between instructions.

Our approach. In this paper, we present LiftFuzz, an approach that uses the instruction context-aware fuzzing technique to create diverse contexts for validating the binary lifters. Specifically, in order to generate test cases that include relationships among instructions, we propose to train an assembly language model, founded on a state-of-the-art machine learning framework, GPT [44], to learn complex interactions among instructions based on control flow and data flow. Upon completion of the training, the assembly language model is then utilized to automatically generate raw assembly test materials. Next, we refine and amalgamate the raw assembly materials to create the assembly test cases, which are then compiled into binary executables. These generated binaries are then fed to binary lifters, which will output translated IRs. Then LiftFuzz proposes an IR integrator to convert the translated IRs into recompilable IRs and further compile them into binary executables. Finally, to validate the lifted IRs, we execute both the recompiled binary executable and the original binary on a physical CPU, extracting their runtime information to compare and detect any inconsistencies.

By utilizing the understanding of interactions among instructions, LiftFuzz substantially outperforms the baseline, requiring only 1/1000 of the test cases used by the baseline to identify 26 inconsistencies, of which 10 have been manually confirmed by the developers. Impressively, this is a 767% increase compared to the inconsistencies detected by the baseline. It shows that accurately lifting an instruction in one context does not guarantee correctness in others. LiftFuzz also finds inconsistencies within instructions can lead to incorrect propagation of effects among instructions and highlights critical issues in binary lifters, such as the lack of high-level information for program structure recovery.

Contribution. This paper makes the following contributions:

- We propose a novel approach to validate binary lifters. Unlike the previous studies that focus on verifying the instructions in isolation, we delve into assessing the binary lifter's ability to handle the intricate interactions among instructions.
- We implement LiftFuzz[1], to our best knowledge, the first research work to utilize the context-aware GPT-based fuzzing technique to validate binary lifters, incorporating an assembly language model to learn interactions among instructions and generate test inputs with contextual knowledge.
- We evaluated LiftFuzz and the baseline on three predominant binary lifters. Experiment results demonstrate that LiftFuzz significantly outperforms the baseline, identifying inconsistencies 767% more effectively with just 1/1000 of the test cases used by the baseline. Notably, LiftFuzz not only unearthed a category of issues that remained undiscovered in prior studies but also enriched the understanding of problems previously identified.

[1]LIFTFUZZ available at https://github.com/zyt755/LIFTFUZZ

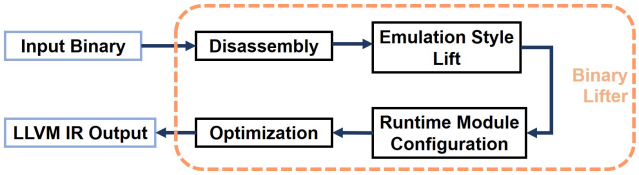

Figure 1: Workflow of Binary Lifters

2 BACKGROUND

Binary lifters are important tools in binary analysis to translate machine code into a higher-level representation, *e.g.*, LLVM IR. The primary goal is to enable the analysis and manipulation of binary code at a higher level of abstraction, which can be more easily understood and modified by researchers and developers. Figure 1 depicts the high-level workflow of binary lifters, which involves several key steps:

- **Step I: Disassembly.** The binary code is loaded into the disassembler, which analyzes the code to determine its architecture, entry point, and function boundary, etc. Then, the disassembler converts the machine code instructions into their corresponding mnemonic representations. To date, the process of disassembling non-obfuscated executables can be executed smoothly and accurately [4, 53].
- **Step II: Emulation Style Lift.** Once the disassembly process is complete, the translation is carried out instruction by instruction, mapping each machine instruction to a sequence of corresponding IRs. The primary objective of these IRs is to accurately emulate machine execution, ensuring the preservation of semantics, memory updates, and other side effects of the original machine code within the lifted IRs. Hence, this step is referred to as emulation-style lifting [32].
- **Step III: Runtime Module Configuration.** When executing machine instructions, the runtime environment undergoes updates, such as modifications to CPU registers, flags, stack, and heap. Consequently, lifted IR code often incorporates specific data structures to represent the runtime environment and aid in computation. For example, in the case of `McSema`, its generated IR code defines a runtime module comprising three elements `(PC, mem, state)`: `PC` denoting the program counter, `mem` representing memory and global data regions, and `state` maintaining registers and CPU flags [50].
- **Step IV: Optimization.** Although the output of the emulation-style lifting process is straightforward, the resulting low-level IR tends to be quite redundant and difficult to comprehend. As illustrated in the Listing 3, a single machine instruction can be translated into several LLVM IRs, often involving many complex operations. To tackle this problem, the binary lifter incorporates many optimization passes to streamline the output IR, making it easier to analyze and

comprehend. However, it is important to note that during the optimization process, the correspondence with the original instructions will be lost. This can result in difficulties when attempting to trace back to the original instruction.

Challenges in Binary Lifting. Despite numerous attempts to improve binary lifters, they continue to grapple with several challenges. Firstly, the vast amount of information within an ISA can be overwhelming, making it difficult to fully understand and encode the effects of countless instructions. Secondly, ensuring the lifted IRs accurately emulate the original instruction is a daunting task, given the need to accommodate a wide array of scenarios during the lifting process. Furthermore, the absence of a definitive reference for the output IRs complicates the development of the binary lifter. Lastly, complex interactions among instructions can bewilder the binary lifter, leading to incorrect functionality. Therefore, verifying the correctness of the binary lifter is crucial to instilling confidence in downstream applications. This underscores the unique value of our efforts in identifying issues in binary lifters.

3 MOTIVATION AND OVERVIEW

3.1 Problem Definition

Given a program P, we denote with $\hat{P}$ a lifted program that emulates P. A state of the program, $s_i \in P$, consists of the program counter pc_i, the state of the CPU registers R_i, the state of the memory M_i, and the execution status E_i. For conciseness, we represent a state of the program P and $\hat{P}$ with the tuple $s_i = (pc_i, R_i, M_i, E_i)$ and $\hat{s}_i = (\hat{pc}_i, \hat{R}_i, \hat{M}_i, \hat{E}_i)$ respectively.

Definition 1. *Let $\hat{P}$ be the lifted program of P. Let S and $\hat{S}$ be the status set of program P and $\hat{P}$, respectively.*

Define that P is consistently lifted to $\hat{P}$ for any $\hat{S}$, iff: $\forall \hat{s}_i \in \hat{P}, \exists s_i \in P$, it holds that:

$$(pc_i = \hat{pc}_i) \& (R_i = \hat{R}_i) \& (M_i = \hat{M}_i) \& (E_i = \hat{E}_i) \quad (1)$$

According to the definition, if there exist $s_i \in S$ and $\hat{s}_i \in \hat{S}$, where their program counters point to different addresses, their states of the register are updated differently, their memory is updated differently, or their execution statuses are different, it indicates the presence of an inconsistency between P and $\hat{P}$. Our objective is to effectively analyze $\hat{P}$ in order to investigate whether there are any inconsistencies with P.

3.2 A Motivation Example

We employ a real-world example from Revng [45] to illustrate our observation, motivation, and the challenges we encounter. In the given example, we defined three distinct contexts for the target instruction `jg` and input them into Revng.

As shown in Listing 1, the code above the dotted line, specifically Line 2 to Line 6, represents the test case's assembly code fed into Revng [45]. On the other hand, the code below the dotted line, i.e., Line 9 to Line 12, represents the output IRs that Revng translated for the instruction in yellow, specifically Line 3. This format is also applicable to the other two scenarios.

Motivated Case Scenario 1. In the first scenario, as shown in Listing 1, after executing the instruction `cmp %rax, 0x40(%rsp)`, the emulated CPU state represents the necessary condition for the `jg` instruction and is stored in the emulated register `@cc_src`. This result can be directly utilized as the jump condition in the IRs for the `jg` instruction.

Table 1: Comparison with closely related works in addressing challenges and used dataset

System	Year	C1	C2	C3	C4	Dataset
AVBT[2] [10]	2015	✗	✗	✓	✗	COTS
MeanDiff [27]	2017	✓	✓	✗	✗	Random Generate
SVBL[3] [12]	2020	✓	✗	✓	✗	Formal Semantic
LifterFuzzer	2023	✓	✓	✓	✓	Customized

[2] [3] are abbreviations of their titles respectively.

Motivated Case Scenario 2. In the second scenario, as shown in Listing 2, the impact of the previous instruction Line 2, `sub $0x8, %r8d` on the emulated `$eflags` register is solely stored in `@cc_src`. Therefore, the IRs for `jg` instruction retrieve data from the `@cc_src` and perform further calculations to determine the jump instruction.

Motivated Case Scenario 3. In the third scenario, as shown in Listing 3, Line 5 `jg` instruction follows Line 4, `lea 0x1(%r14), %eax` instruction, which does not have any impact on the emulated `$eflags` register. Therefore, its IRs do not modify related variables representing `$eflags`. In order to retrieve the required flag bits, the IRs for `jg` instruction have to load data from emulated registers, `@cc_op`, `@cc_src`, `@cc_src2` and `@cc_dst` stored by Line 2, `cmp %rax,0x40(%rsp)` and then perform immediate calculations based on these data.

As illustrated in the above three scenarios, Revng generates three unique IRs for the `jg` instruction due to varying contexts. This underscores that validation of each instruction in isolation falls short of a comprehensive assessment. Thereby, it is essential to verify the binary lifter's capability to manage the interactions between instructions.

3.3 Challenges

C1: Diverse range of test instruction inputs. The vast number of instructions within an ISA makes it difficult to create test cases that cover all possible scenarios. Different opcodes and operands can be combined into numerous possibilities.

C2: Variety of output IR. As demonstrated in the previous Section 3.2, the binary lifter may generate multiple variations of IRs for a single instruction. Furthermore, different binary lifters may also produce different IRs for the same instruction.

C3: Absence of ground truth for output IR. Since the output of binary lifters is low-level IRs, it differs markedly from the high-level IRs generated by compilers. As a result, there is no direct correspondence between the two, making it challenging to establish a ground truth for direct comparison.

C4: Sophisticated interactions among instruction. Constructing context-aware test inputs is necessary to validate whether the

```
/* Assembly codes */
0x40050f:    cmp     %rax,  0x40(%rsp)
0x400514:    jg      40051b <Block_3>
0x400516:    cmp     %rax,  0x40(%rsp)
0x000000000040051b <Block_3>:
0x40051b:    cmp     %rax,  0x40(%rsp)
------------------------------------------------
/* LLVM IR */
call void (ptr, ...) @newpc(...)
%158 = load i64, ptr @cc_src, align 8
%159 = icmp sgt i64 %156, %158
br i1 %159, label ..., label ...
```

Listing 1: Motivated Case Scenario 1: Revng lifted Line 3 `jg` instruction to IRs span from Line 9 to Line 12.

```
/* Assembly codes */
0x400500:    sub     $0x8,  %r8d
0x400504:    jg      400506 <Block_1>
0x0000000000400506 <Block_1>:
0x400506:    cmp     $0x47, %r8b
------------------------------------------------
/* LLVM IR */
call void (ptr, ...) @newpc(...)
%sext578 = shl i64 %131, 32
%134 = load i64, ptr @cc_src, align 8
%sext579 = shl i64 %134, 32
%135 = icmp sgt i64 %sext578, %sext579
br i1 %i135, label ..., label ...
```

Listing 2: Motivated Case Scenario 2: Revng lifted Line 3 `jg` instruction to IRs span from Line 8 to Line 13.

```
/* Assembly codes */
0x4004f4:    cmpb    $0x5f,       0x2(%rbp)
0x4004f8:    jg      400500 <Block_1>
0x4004fa:    lea     0x1(%r14),   %eax
0x4004f3:    jg      400500 <Block_1>
0x0000000000400500 <Block_1>:
0x400500:    cmpb    $0x5f,       0x2(%rbp)
------------------------------------------------
/* LLVM IR */
call void (ptr, ...) @newpc(...)
%148 = load i64, ptr @cc_dst, align 8
%149 = load i64, ptr @cc_src, align 8
%150 = load i64, ptr @cc_src2, align 8
%151 = load i32, ptr @cc_op, align 4
%152 = call i64 @helper_cc_compute_all(...)
store i64 %152, ptr @cc_src, align 8
store i32 1, ptr @cc_op, align 4
%153 = lshr i64 %152, 4
%154 = xor i64 %153, %152
%155 = and i64 %154, 192
%156 = icmp eq i64 %155, 0
br i1 %156, label ..., label ...
```

Listing 3: Motivated Case Scenario 3: Revng lifted Line 5 `jg` instruction to IRs span from Line 10 to Line 22.

Figure 2: Motivation cases that show the same assembly instruction with different contexts are lifted into different IRs.

binary lifter can handle interactions among instructions. Nonetheless, extracting and utilizing these complex interactions for testing purposes can be quite difficult.

3.4 Prior Efforts and Our Insights

As shown in Table 1, prior works like [10, 12, 27] only try to address the **C1**, **C2**, and **C3**, without fully considering **C4** and ignoring interactions between the instructions. Chen et al. [10] directly utilize a pre-existing data set, EEMBC 1.1 benchmark [16], to evaluate their validator. Unfortunately, this dataset is not designed for validating the binary lifter. Kim et al. [27] employ the symbolic summary method to streamline randomly generated test cases and uphold coverage. Dasgupta et al. [12] leverage the formal semantics of x86 instruction and LLVM IR to achieve comprehensive coverage. They have assumed that the binary lifter processes instructions without considering the corresponding contexts, consistently generating identical IRs for each identical instruction. This results in each instruction being validated in isolation, thereby overlooking the potential interactions among instructions. However, such an assumption is wrong, as we observed that the binary lifter can produce different IRs for the same instruction with different contexts.

Based on our observation, we highlight the inadequacy of solely validating individual instructions and stress the importance of verifying the binary lifter's capability to handle interactions among instruction contexts accurately. *The primary focus is on creating test cases that not only ensure the syntactic correctness of the test instructions but also encapsulate the intricate interrelationships among these instructions.* The previous rule-based generator in [27], while ensuring the correctness of instruction syntax, did not sufficiently incorporate diverse interactions among instructions and did not consider test case mutations. To mitigate this issue, we employ a GPT-based assembly language model. The ability of GPT [44] to understand complex patterns is crucial for producing instructions that are not only syntactically and semantically precise, but also encompass intricate interconnections. Simultaneously, we provide varied prompts to the assembly language model to produce ample test case mutations. This use of GPT allows LiftFuzz to generate high-quality and diverse test cases that meet comprehensive testing requirements.

3.5 Overview

Figure 3 presents the overview of LiftFuzz, which consists of three main components: the test case generator, the IR integrator, and the validator. The test case generator is designed to produce test assembly codes and compile them as test inputs for fuzzing tests. Once the test inputs are obtained and fed into the binary lifter, the IR integrator is employed to ensure that the output IRs of the binary lifter can be recompiled. Afterwards, the IR integrator carries out instrumentation for subsequent verification and backtracking purposes. Lastly, the validator is utilized to verify the consistency between the instructions on the physical CPU and their corresponding IRs, generating reports accordingly. LiftFuzz generally applies to verify binary lifters from any ISA, *e.g.*, x86, ARM, RISC-V, MIPS,

PowerPC, to an intermediate representation, such as LLVM IR. In Section 4, we will introduce each component of LiftFuzz.

4 DESIGN OF LIFTFUZZ

4.1 Test Case Generator

As previously mentioned, the preliminary step for utilizing LiftFuzz requires using the binary executables as test inputs to fuzz the binary lifter. Consequently, LiftFuzz incorporates a test case generator designed to produce context-based test cases that encapsulate the interactions among instructions. The generator's workflow is depicted in the Figure 4, with context extraction serving as the initial phase. As outlined in Section 4.1.1 and Section 4.1.2, we observe that the code context information is naturally embedded inside the Control Flow Graph (CFG) and Data Flow Graph (DFG), so we extract the instruction sequences that bear the control flow and data flow information to construct our dataset. This dataset is then split into two parts, each serving a unique function. One part is used for model training (Section 4.1.3), where the sequences of instructions are segmented into tokens and then fed into the GPT model, enabling the model to learn the context information among instructions and subsequently generate test cases based on this knowledge. Once the model's training is complete, the remaining instruction sequences serve as seed input, providing the fully trained model with varying initial contexts for test generation, as elaborated in Section 4.1.4.

4.1.1 Collection of Interactions Between Instructions. As mentioned in Section 3, it is crucial to verify the ability of binary lifters to accurately handle the interactions between instructions. However, it should be noticed that instructions within a program interact with each other in a multitude of ways. Here are some of the key interactions between instructions:

Data dependencies. Instructions can have dependencies on the data produced by previous instructions. For instance, if instruction A writes a value to a register and instruction B subsequently reads from the same register, instruction B relies on the execution result of instruction A. Proper management of these dependencies is crucial to ensure accurate execution.

Control flow. Instructions have the ability to modify the execution flow within a program. For instance, conditional branch instructions like `jz`/`jnz` or `je`/`jne` can alter the sequence of instructions execution based on specific conditions.

Memory access. Instructions can read from or write to memory locations, enabling data storage, retrieval, or modification. Proper synchronization is necessary when multiple instructions access the same memory location to prevent conflicts.

Instruction sequencing. Instructions are executed sequentially by the processor, following the program's control flow. The order in which instructions are executed can affect the program's behavior and the states of registers and memory.

Instruction dependencies. Some instructions have specific ordering requirements. For example, if an instruction modifies a flag, subsequent instructions that depend on the state of the flag should be executed after it. Similarly, instructions that modify the memory should ensure that the memory is in a consistent state before other instructions access it.

To our best knowledge, none of the prior works [10, 12, 27] has fully considered these interactions, which means the existing methods did not effectively meet the goals that comprehensively validate the binary lifter. To this end, LiftFuzz has utilized an assembly language model to collect interactions among instructions.

4.1.2 Context Extraction. The goal of context extraction is to construct a dataset consisting of varying contexts, which not only significantly boosts the model's ability to understand the relationships among instructions but also provides the model with diverse initial contexts for generating test cases. As previously mentioned in Section 4.1.1, we have observed that the code context information is naturally embedded within structures such as the CFG and DFG. Based on this observation, we first disassemble binaries and extract def-use relations using Binary Ninja [52]. Next, we analyze dependencies that exist among registers, memory locations, and function call arguments. Additionally, we consider the implicit dependencies introduced by `eflags`. Then, we obtain each operand's data dependencies and establish def-use relationships between instructions and their respective dependent instructions. Subsequently, we extract instruction sequences from control flow sequences as well as sequences guided by def-use relationships to construct our dataset. Finally, the dataset is partitioned into two parts, each fulfilling a distinct purpose. One part is used to train the model as elaborated in Section 4.1.3. The other part acts as seed input to supply the well-trained model with initial varying contexts for test case generation, as described in Section 4.1.4.

4.1.3 Design of Assembly Generator Model. As mentioned in Section 4.1.1, previous approaches have proven ineffective in generating context-aware test cases. Additionally, to improve the efficiency of fuzzing testing, it is crucial for test cases to include an adequate amount of mutations and variations, rather than strictly adhering to normal instruction relationships. In order to address these challenges, we have shifted our attention to machine learning methods. After thorough consideration, we have chosen to base the assembly generator model of LiftFuzz on GPT [44], incorporating the following significant design considerations:

Tokenization. In order to augment the deep neural network's comprehension of the internal structures within the instruction, LiftFuzz integrates two methodologies: position embedding and a training task from the field of Natural Language Processing (NLP) known as Causal Language Modeling (CLM) [43]. Position embedding equips our assembly language model with information about the placement of basic tokens in an instruction, a crucial aspect because the order of tokens can alter the interpretation of an instruction. The CLM task facilitates the training of our assembly model to understand the sequential relationships and dependencies inherent in the instructions.

To capture the intricate internal formats of instructions, we employ a fine-grained strategy that involves decomposing each instruction into its constituent parts. This approach allows the model to analyze and understand the structure of instructions at a granular level. Inspired by PALMTREE [31], we treat each instruction as a discrete sentence, breaking down the instruction into its fundamental

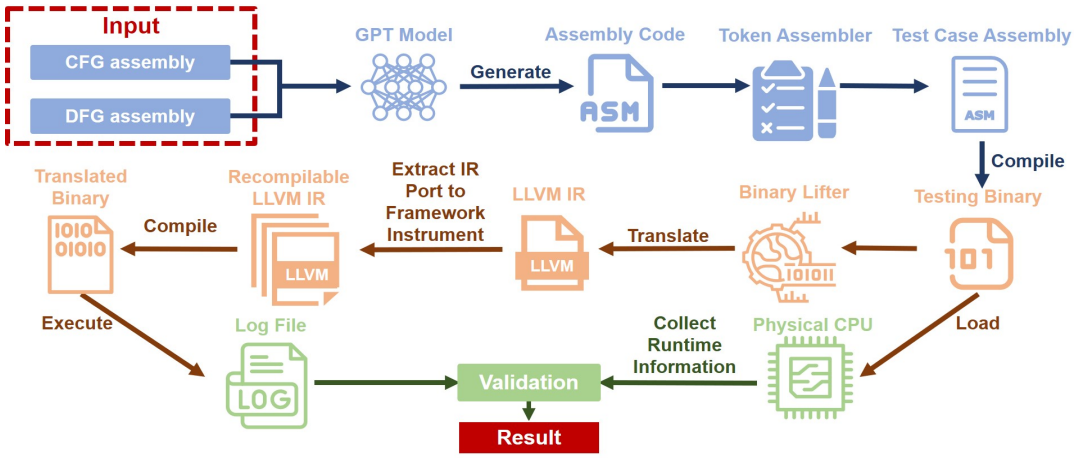

Figure 3: Overview of LiftFuzz

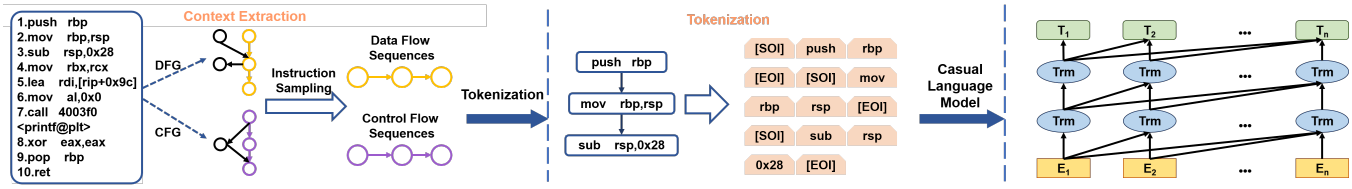

Figure 4: Workflow of test case generator in LiftFuzz

components, encompassing opcodes, registers, intermediate numbers, strings, and symbols, among others. For example, given the instruction, `lea rbp,[rsp+0x70]`, we dissect it into the following elements: "**lea**", "**rbp**", "**[**", "**rsp**", "**+**", "**0x70**", and "**]**".

LiftFuzz employs a specific normalization strategy to mitigate the Out-Of-Vocabulary (OOV) issue induced by strings and constant numbers. The special token **[string]** is utilized to substitute strings. For constant numbers, it is necessary to ascertain whether it represents an address or a value. If it is an address, the precise value is not particularly beneficial for our models and can be replaced with the **[address]** token. Conversely, if it is a value, it might contain vital information pertaining to accessed local variables, function arguments, and data structure fields. Therefore, LiftFuzz retains these as tokens and encodes them using one-hot vectors. Given that LiftFuzz is founded on GPT, we have incorporated additional tokens, **[SOI]** and **[EOI]**, to enhance the model's learning capacity. Here, **[SOI]** signifies the beginning of an instruction, while **[EOI]** denotes its end.

Assembly Language Model. Moreover, we would like to train the assembly language model to capture the relationships among instructions. Given that GPT handles tokens from left to right and lacks a clear concept of sentences, it becomes challenging for GPT to capture the relationships between instructions or comprehend the boundaries between them. To address this limitation, inspired by Next Sentence Prediction (NSP) [14], we preprocess the training data by following BERT's method. This involves clearly dividing each instruction and incorporating the concept of the instruction into the input. By introducing explicit instruction boundaries, our model can potentially capture the relationships among instructions and improve its understanding of the input structure. Furthermore, unlike natural language, the semantics, syntax, usage, and order of instructions in programming languages are strictly specified and cannot be altered arbitrarily. For example, the `jmp` instruction must be followed by a legal address symbol; otherwise, an error will occur. Consequently, the def-use relation among instructions is clearly defined and remains unchanged even with varying compiler optimizations. Leveraging these characteristics, we incorporate data dependencies into the training data. In this way, we enhance our model's language generation capabilities while also enabling it to learn the relationships among instructions in a manner similar to BERT's NSP task.

As depicted in Figure 5, the initial token of this amalgamated input is an unique token, denoted as **[SOI]**, while the final token is represented as **[EOI]**. Subsequently, we incorporate the position embedding into the token embedding, using this blended vector as the input for the transformer network. These enhance the deep neural network's understanding of the inherent structures within the instruction.

Causal Instruction Modeling. The task we employ to pre-train the assembly language model, which we call Causal Instruction Modeling (CIM), is based on Causal Language Modeling (CLM). The objective of CIM is to predict the subsequent token in an instruction, given the preceding words. Given an instruction, the notation t_i represents a token in a sequence of instructions $I = t_1, ..., t_n$. The transformer decoder in the model is trained to predict the next tokens in the sequence. It outputs a probability for each possible token

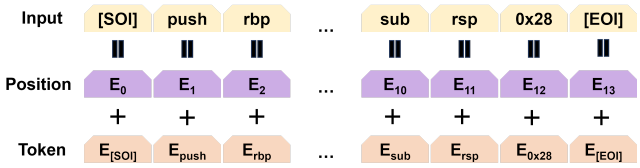

Figure 5: Input representation

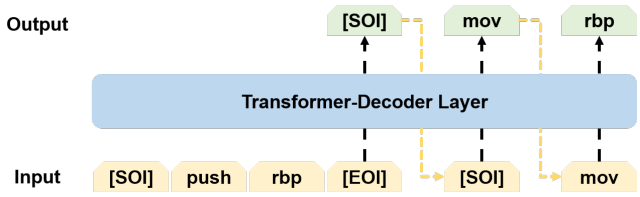

Figure 6: Prediction representation

t_i using a softmax layer located at the top of the transformer network. We use a standard language modeling objective to maximize the following likelihood:

$$\mathcal{L}(I) = \sum_{i=1}^{|I|} (logP(t_i|t_{i-k}, ..., t_{i-1}; \Theta)) \tag{2}$$

where k is the size of the context window and the conditional probability P is modeled using a neural network with parameters θ. These parameters are trained using AdamW [33].

$$U = (t_{i-k}, ..., t_{i-1}) \tag{3}$$

$$h_0 = UW_e + W_p \tag{4}$$

$$h_l = F_S(h_{l-1}) \forall l \in [1, n] \tag{5}$$

$$P(u_i|u_{i-k}, ..., u_{i-1}; \Theta) = \frac{exp(h_n W_e(u_i)^T)}{\sum_{t=0}^{N} exp(h_n W_e(u_t)^T)} \tag{6}$$

where U is the context vector of tokens, n is the number of layers, W_e is the token embedding matrix, W_p is the position embedding matrix, $F_S(\cdot)$ is the function that leverages the algorithm in the position-wise feed-forward network together with self-attention to calculate the hidden vector h_l, and N is the number of tokens.

Figure 6 illustrates how the above formulas guide the model to learn the inherent structures within the instructions, as well as the interactions among them. Initially, the model is fed with "**[SOI] push rbp [EOI]**". In order to correctly predict the next token, the model need to understand the meaning and usage of **[SOI]** and **[EOI]**, recognizing that "**[SOI] push rbp [EOI]**" constitutes a complete instruction. This understanding allows the model to anticipate that the next token should be the start of a new instruction, thereby correctly predicting that the next token will be **[SOI]**.

In the following step, the token **[SOI]** is integrated with the preceding instruction sequence to form a new sequence, "**[SOI] push rbp [EOI] [SOI]**", which is re-input into the model. To predict the subsequent token, the opcode of the next instruction, the model must comprehend **mov** is the opcode of instruction and the interactions between **push** and **mov** instructions. Based on this understanding, the model predicts that the next token will be **mov**.

Subsequently, the token **mov** is combined with the previous instruction sequence and re-input into the model. Based on the understanding of the inherent structures within the **mov** instruction and interactions between **push rbp** and **mov rbp rsp**, the model then predicts the next token to be **rbp**.

This cycle of predicting tokens, integrating them with the previous instruction sequence and re-inputting them into the model, continues until an instruction sequence of the specified length is generated. Each generation demands that the model accurately learns the inherent structures within the instruction and interactions among instructions. Failing to do so, the model could be overwhelmed by a plethora of alternative tokens, leading it to make incorrect selections. Under such circumstances, the model would receive negative feedback, prompting it to adjust its parameters until it is capable of accurately predicting the subsequent token. Hence, upon completion of the model training, the model has effectively learned the inherent structures within the instruction as well as interactions among instructions.

4.1.4 Binary Executable Generation. After training the assembly model, we generate test data without using traditional input mutation methods like random mutation, combination, and permutation. These methods can inevitably disrupt the interactions among the model-generated instructions. Instead, we employ two distinct methods to incorporate sufficient variation in the test cases.

Firstly, we provide the model with varying contexts. As discussed in Section 4.1.2, we construct our dataset by extracting instruction sequences that carry control-flow and data-flow information from CFGs and DFGs. This data is then utilized as seed input to provide the model with diverse initial contexts. For instance, as depicted in the Listing 1, Listing 2, and Listing 3, the target instruction `jg` is tested within diverse contexts. This approach enables the generation of varied test cases, facilitating the multiple execution of the testing instruction rather than merely testing the same instruction once, as was the practice in prior work [10, 12]. Secondly, we enable the model to continuously generate as many instructions as possible. By capitalizing on the characteristics of the Attention mechanism [51] and the model's diversity, we observe that an increase in the number of generated instructions correspondingly enriches the context among these instructions. Additionally, hallucinations are allowed and can be regarded as mutations in the test input. Compared to traditional random input mutation, these methods not only eliminate a multitude of meaningless test cases, but also supply a diverse range of context information, which is crucial for validating the binary lifters.

Token Assembler. Since the model-generated tokens are not interpretable to the compiler, the Token Assembler is introduced for syntax-checking and organizing tokens into instructions that will later be handled by the compiler properly. The Token Assembler functions through three primary operations. The first one involves removing tokens that are helpful in training the model while redundant in the compilation. The second one is dedicated to adding punctuation, whereas the third one is aimed at eliminating syntax errors. For example, when the output sequence "**[SOI] push rbp**

[EOI] [SOI] mov rbp" depicted in Figure 6 is fed into the Token Assembler, it initially removes tokens such as **[SOI]** and **[EOI]**, which are meaningless to the compiler. Subsequently, the Token Assembler incorporates "**\n**" to split the sequence into two instructions. Ultimately, the Token Assembler conducts syntax-checking and discards the tokens **mov** and **rbp** due to their inability to construct an incomplete instruction.

It is noteworthy that the diverse contexts inside seed inputs and the contextual knowledge that the model learned from the training dataset are sufficient for LiftFuzz to generate high-quality test cases. Hence, the Token Assembler's sole contribution is to ensure the smooth compilation of tokens into binary executables without directly affecting the overall performance.

The binary executables serve two unique purposes. Firstly, it is sent to IR Integrator as test data, discussed in Section 4.2. Secondly, it is sent to the Validator as ground truth, elaborated in Section 4.3. Unlike previous work [12, 27] that translated the emulation-style IR into another IR for comparison, we conducted our unification in the same IR as the original binary lifter. This method preserves the semantics and usage of the emulation-style IR, thereby eliminating potential errors that may be introduced during translation.

4.2 IR Integrator

4.2.1 Converting IR to Recompilable. After obtaining binary executables, LiftFuzz feeds them into the binary lifter, which initially generates emulation-style IRs. These IRs are then optimized into the final IRs. Hence, the accuracy of the initial emulation-style IRs directly affects the quality of the final output, making it significant. In addition, unlike the optimized IRs, the emulation-style IRs not only maintain a correlation with the original instruction but also preserve the semantics. This makes them particularly useful for comparison testing. Based on these, we select the emulation-style IRs from the output of the binary lifter for the verification purpose.

Unfortunately, not all emulation-style IRs can be successfully compiled and executed. Simultaneously, various binary lifters produce different styles of IRs, which need to be unified before comparison. Besides, we also need to instrument the emulation-style IRs to extract the runtime state, confirm its accuracy, and pinpoint any incorrect IRs. As a result, there exists a need for uniform processing of the emulation-style IRs produced by different binary lifters. To achieve this, LiftFuzz provides an IR framework to ensure that the emulation-style IRs can be recompiled and run smoothly. This framework has made the following preparations. First, it unifies different representations of machine states from various IRs. Next, it provides runtime modules that support the emulation-style IRs. Then, it extracts and merges the related dependencies of different IRs into the recompilable file. Lastly, it instruments the IRs for validation and backtracking purposes.

Machine State. Since different binary lifters might utilize diverse data types and different data structures to depict the same physical CPU register, standardizing the different machine state representations produced by various binary lifters into a single format is necessary. Furthermore, different binary lifters have distinct initialization methods. Therefore, to guarantee the consistency of their initial states, the emulation-style IRs must be subjected to a uniform initialization process.

Algorithm 1: Overview of Validator

```
Input: Log file L
       Binary executable B
Output: Inconsistencies 𝕀
Initialize the ground truth GT with L
for record ∈ L do
    GT.next();
    if GT.address ≠ record.address then
        Find different execution path p
        Append p to 𝕀
    end
    if GT.CPU_state ≠ record.CPU_state then
        Find different CPU state C_s
        Append C_s to 𝕀
    end
    if GT.memory ≠ record.memory then
        Find different memory state M_s
        Append M_s to 𝕀
    end
end
if GT.exit_state ≠ record.exit_state then
    Find different final exit status E_s
    Append E_s to 𝕀
end
```

Runtime Module. All operations of the emulation-style IRs simulate real instructions. When executing real instructions, updates occur in the runtime environment, such as registers, flags, and stack operations. Hence, to ensure the smooth recompilation and execution of the emulation-style IRs, the IR framework replicates an environment similar to a real physical CPU. Besides, due to these modifications, runtime functions, *e.g.*, `@printf()` of GLIBC [18], also require to be rewritten to ensure that the program has the necessary resources and environment to run smoothly. All these processes must be transparent to the emulation-style IRs, ensuring the verification is accurate and does not introduce new errors.

Related Dependency. Each binary lifter has its own unique methods of representation and calculating intermediate states, *e.g.*, the `@helper_cc_compute_all()` function of `Revng`. The accurate functioning of their IRs relies heavily on these functions and variables. We will not alter this part of the emulation-style IRs, but rather directly transplant them into the recompilable file. Let us take the `@helper_cc_compute_all()` function as an example. Since this function is responsible for calculating the state of `eflags`, we employ a probe to directly output its return value whenever this function is invoked. Subsequently, we compare this value with the physical `eflags`. This approach keeps our framework lightweight and guarantees that no new errors will be introduced during the integration process.

Instrumentation. The final step in processing emulation-style IRs involves inserting the instrumentation code into each instruction's corresponding IRs. Instrumentation serves two purposes. Firstly, it outputs the machine state after the execution of each instruction's

corresponding IRs, verifying whether IRs faithfully simulate the instruction. Secondly, it establishes a mapping relationship between the instruction and corresponding IRs. This can act as a point of reference for backtracking when inconsistencies are later identified.

Upon completing the above steps, the emulation-style IRs can be successfully compiled and executed. Thanks to the instrumentation, when the recompiled file is run, the file simultaneously outputs a log file that records the machine state after the execution of each instruction's IR block. This log file is then sent to the validator, introduced in the following section, for verification.

4.3 Validator

As depicted in Figure 3, the log file from the IR integrator serves as the verification object, and all of the original binary executables are sent to the validator for comparison and verification. Unlike previous work [10] that relied on an emulator as the ground truth, LiftFuzz adopts a more reliable method. We utilize the physical CPU as an oracle. Since the corresponding machine state after executing IRs of each instruction has been recorded in the log file, we only need to run the original binary executable on the physical CPU to extract the runtime machine state as the ground truth. We then compare the ground truth with the records in the log file one by one to identify any inconsistency. This method eliminates errors introduced by the emulator and streamlines the entire verification process, making it more lightweight.

The algorithm 1 explains the algorithm of the validator. The validator takes both the log file and the original binary executable as inputs. Then, the validator loads the original binary into the CPU. Subsequently, the validator initializes the machine state of the binary executable to match that of the log file. Finally, the validator executes one instruction of the original binary, extracts the machine state as the ground truth, and compares it with the corresponding record in the log file. If the address of the log record differs from the ground truth, it indicates an incorrect execution path in the emulation-style IRs. Similarly, if the machine state, or the final execution status of the log record differs from the ground truth, it suggests that the emulation-style IRs do not faithfully simulate the original instruction.

5 EVALUATION

In this section, we present the experimental evaluation of LiftFuzz. We aim to address four questions through these experiments:

RQ1. Can LiftFuzz learn interactions among instructions and then generate test cases with various contexts?

RQ2. Can LiftFuzz identify any unknown inconsistencies resulting from interactions among instructions?

RQ3. Can LiftFuzz find inconsistencies inside a single instruction?

RQ4. Can LiftFuzz find the root cause of the inconsistencies between the assembly instructions?

Implementation. LiftFuzz is implemented with over 5,500 lines of code, incorporating both Python and LLVM IR. The test case generator in LiftFuzz is built upon `minGPT` [26] and `PALMTREE` [31]. The reported results are achieved in a setting with a learning rate of 5e-4, a dropout rate of 0.1, and a batch size of 64. The system also incorporates 8 transformer decoder layers with 16 self-attention heads and 512-dimensional hidden states.

Table 2: Evaluation against LSTM and NMT

Model	Perplexity	BLEU	Self-BLEU
LSTM	25.3306	48.51	20.87
NMT	13.8120	65.18	90.33
GPT	15.0192	62.49	25.25

Dataset. To implement the test case generator, we collected a total of 16 open-source projects to provide training data for assembly language model training, and these projects are widely used in recent five years of reverse engineering research works [15, 25, 40, 41]. Specifically, the collected projects are relevant to utilities (`Attr`, `Bash`, `Binutils`, `Coreutils`, `Diffutils`, `Findutils`, `Make`, `Sg3_utils`, `Tar`), database management (`Sqlite3`), and networking (`Httpd`, `Openssh`, `Openssl`, `Putty`, `Tmux`, `Wget`). Finally, our training dataset collects more than 200 million instructions as the starting point for assembly language model training.

Evaluation Environment. The training of the assembly language model and test cases generation are performed on a server with Intel Core i7-7820X v8 CPU @ 3.60GHz, 64 GB of memory, and 2 NVIDIA GeForce GTX 1080 Ti GPUs, running on Debian 12. To facilitate a paralleled evaluation process for the generated test cases, we employed an additional server equipped with an AMD EPYC 7543 v128 CPU @ 2.80 GHz and a substantial 512 GB of memory, running on Ubuntu 22.04.3 LTS.

5.1 RQ1: Context-based Test Case Generation

To address RQ1, we scrutinize LiftFuzz from three unique standpoints. Initially, we evaluate the GPT model and other sequence-based models, aiming to gauge the model's ability to produce diverse and high-quality test cases. Going further, we aim to ascertain if alterations in context can notably affect the model's quality and diversity, thereby impacting the generation of high-quality test cases. We thus implement varying block sizes to limit the model's access to contextual information to achieve this. Finally, we conduct a comprehensive experiment to evaluate how well the model contributes to the final results. Both LiftFuzz and the baseline are evaluated using three state-of-the-art binary lifters.

5.1.1 Evaluation with Other Sequence-based Models. To evaluate the model's ability to produce diverse and high-quality test cases, we carefully chose two sequence-based models, LSTM and NMT, effective at text generation tasks. We then compare these with the GPT models, utilizing three standard and comprehensive metrics.

Other Sequences-based Models. Long Short-Term Memory (LSTM) [22] is a variant of the recurrent neural network (RNN) designed to address the vanishing and exploding gradients issue common in conventional RNNs, thereby improving efficiency with longer sequences. Due to its ability to learn long-term dependencies, which is crucial in text generation as the relevance of words often depends on the context established by preceding words or sentences, LSTM is often used in text generation tasks.

Neural Machine Translation (NMT) [2] is a type of sequence-to-sequence models that use an RNN or a transformer network to process the input sentence and generate the translated sentence.

NMTs are especially useful in text generation tasks because they can generate more fluent and natural-sounding text compared to traditional rule-based or statistical methods.

Implementation. All models use default settings and parameters. Meanwhile, all models are the same in batch size, similar in size, trained with the same dataset, and adopt the same data processing strategy, including tokenization. All models are still underfitting.

Metrics. Perplexity [24] is the standard metric in the language model used to gauge how well a probability model predicts a sample. Perplexity can reflect the diversity of a language model because it essentially measures the uncertainty of a model in predicting the next word in a sequence [48]. Thus, a lower perplexity score indicates that the model has less uncertainty, meaning it can predict the next word more accurately and have more diversity. Generally, a perplexity value under 20 suggests that the model exhibits a high degree of diversity [44].

BLEU (Bilingual Evaluation Understudy) [39] is a popular metric used to evaluate the quality of machine-generated text by comparing it to reference texts. A higher BLEU score indicates a higher quality generation that is more similar to reference texts [30]. BLEU score above 60 indicates that the model has very high quality [20].

Self-BLEU [56, 57] is a variant of the BLEU score that is used to measure the diversity of the generated text. Instead of comparing the generated text to a reference text, Self-BLEU compares each generated sentence to all other generated sentences. A lower Self-BLEU score indicates higher diversity, indicating less similarity among generated sentences. Self-BLEU score below 30 denotes moderate diversity in the model's output [20].

Results. As the table 2 clearly illustrates, LSTM underperforms in terms of the BLEU score, suggesting its inability to generate high-quality test cases. NMT, on the other hand, scores low on the Self-BLEU score, indicating its lack of capacity to generate highly diverse test cases. Contrastingly, GPT excels over both LSTM and NMT in the assembly generation task, displaying a low perplexity of 15.0192 (less than 20) and Self-BLEU scores of 25.25 (less than 30), while simultaneously securing a superior BLEU score of 62.49 (greater than 60). Based on related NLP studies [20, 44, 48], these findings indicate that our assembly language model is capable of generating high-quality test cases with considerable diversity.

These findings align with previous studies in NLP [2, 35, 43, 44, 47, 51], which pinpoint that GPT outshines LSTM and NMT in text-generation tasks. This superiority is primarily due to the GPT model's extensive context window, enabling it to preserve long-term dependencies within the text. This capability proves beneficial in generating text that is both coherent and contextually appropriate. We believe the fundamental similarity between assembly code and natural language underpins this outcome. As a result, the assembly and text generation tasks share essential characteristics, allowing for the transfer of insights and techniques from NLP to the generation of diversity and high-quality assembly code.

5.1.2 Evaluation with Different Contexts. To determine whether altering the context can substantially boost the model's quality and diversity, thus enabling the creation of superior test cases, we impose different block sizes to confine the model's access to contextual information.

Table 3: Evaluation against 3 state-of-the-art binary lifters

Dataset		MeanDiff	LiftFuzz
# Test cases		97,487	1,000
Mcsema	# Lift	94,025 (96.45%)	1,000 (100%)
	# Recompile	93,654 (96.07%)	1,000 (100%)
	# Inconsistencies	**1**	**5**
Revng	# Lift	44,913 (46.07%)	996 (99.60%)
	# Recompile	29,460 (30.21%)	867 (86.70%)
	# Inconsistencies	**1**	**5**
Retdec	# Lift	49,322 (50.59%)	993 (99.30%)
	# Recompile	29383 (30.14%)	924 (92.40%)
	# Inconsistencies	**1**	**16**
# Total found inconsistencies		**3**	**26**
# Context found inconsistencies		**0**	**11**

Implementation. We have set five block sizes (10, 16, 32, 64, and 128) to limit the model's access to contextual information. We start with a block size of 10 to ensure the context includes at least one complete instruction rather than an incomplete one. The maximum block size is set to 128, which is the highest capacity supported by the NVIDIA GeForce GTX 1080 Ti GPU. Simultaneously, all model parameters remain consistent. All models are still underfitting.

Results. As depicted in Figure 7, with an increase in block size, there is a gradual decrease in both perplexity and self-BLEU, indicating a negative correlation. This suggests that the model has less uncertainty and higher diversity, meaning it can predict the next token more accurately, and the generated instructions are less similar to each other. On the other hand, the BLEU score shows a gradual increase, implying a positive correlation. This indicates a higher quality of generation that aligns more closely with reference texts. All scores achieve a marginal effect when the block size reaches 128. These results demonstrate that expanding varying contexts can significantly enhance the model's quality and diversity, thus facilitating it to generate high-quality test cases.

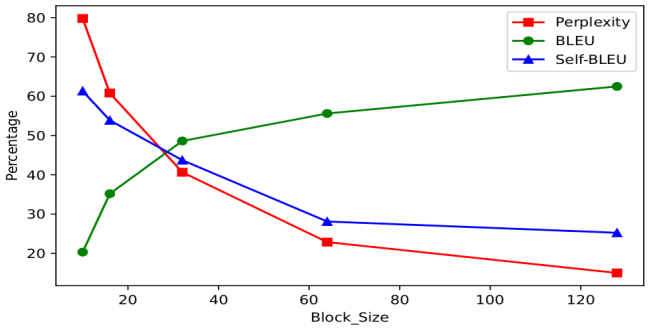

Figure 7: Evaluation with Different Contexts.

5.1.3 Comparison Against MeanDiff. To assess whether LiftFuzz can learn the interactions among instructions and to quantify the model's contribution to the final results, we evaluate both LiftFuzz and the baseline using three state-of-the-art binary lifters.

Baseline. While there is prior work on binary lifter (see also Table 1), AVBT [10] does not make their system publicly available, and SVBL [12] cannot generate test cases since they utilize the existing formal semantics of x86 instruction and LLVM IR for the binary lifter validation. As a result, we use `MeanDiff` [27] as our baseline.

`MeanDiff` [27] checks the semantics equivalence of each instruction returned from a test case generator, a procedure akin to that of LiftFuzz. As such, the test instructions generated by `MeanDiff` serve as a suitable baseline for comparison when assessing the performance of LiftFuzz. Nonetheless, `MeanDiff` only guarantees the correctness of the instruction syntax and does not consider the semantics of instructions or the interactions among them. This simplifies the generation process and facilitates the generation of more test instructions for a wide coverage, yet simultaneously produces an excess of test instructions that are lack of the necessary running environment. Thereby, test instructions generated by `MeanDiff` cannot be directly executed. To execute these instructions, we classify these instructions based on both opcode type and operand type. For instance, we group `add al,0x0`, `add al, 0x42`, and `add al, 0xff` together, then use a template same as LiftFuzz to initialize and finalize the running environment. Subsequently, we compile these grouped instructions into binary executable files.

Evaluation. During the same period, LiftFuzz produced 1,000 test cases while `MeanDiff` generated 97,487. This discrepancy is because LiftFuzz requires more computational power for prediction, whereas `MeanDiff` generates test cases based on simple rules. To ensure every test assembly instruction is compiled into binaries and prevent the compilers from optimizing away the test instructions that are useful in finding contextual bugs but may be meaningless to compilers, we compile *all* test cases with the `-O0` parameter. Additionally, *all* binary executable inputs contain debug information to assist the disassembler in performing accurate disassembly.

We evaluate LiftFuzz and `MeanDiff` on three predominant and mature binary lifters, `RetDec` (v4.0-317-ge59b1388) [1], `McSema` (v3.0.26) [50], and `Revng` [45][4]. `McSema`, `Revng`, and `RetDec` are well-established and widely utilized binary lifters that have undergone rigorous testing and enjoy a strong reputation in the community. Their GitHub repositories indicate a history of approximately seven years, during which their developers have actively maintained them. As a result, it is challenging to find any inconsistency from them.

Results. As shown in Table 3, compared to `MeanDiff`, LiftFuzz generates a higher percentage of test cases that the binary lifter can process, and a larger portion of the lifted IRs can be recompiled back into binary executables, which is especially evident in the validation of `Revng` and `RetDec`. This is primarily because `MeanDiff` evaluates instructions individually and focuses on the semantic equivalence of a single instruction, thereby generating testing instructions independently. Consequently, even though the binary lifter can handle a single binary instruction, it encounters difficulties when processing the testing instruction within a binary executable file, which does not comply with a normal program's basic structure and semantics. Thereby, the binary lifter deals with the test input as an exception and generates IRs containing either pseudocode or statements that have not been implemented, making it unfeasible for recompilation. Since lifting and recompiling are both time-consuming and resource-intensive, considering relationships among instructions allows LiftFuzz to exhibit superior performance efficiency compared to `MeanDiff`.

While `MeanDiff` can generate a large number of test cases, they are not context-aware but simply a combination of single instructions. As a result, despite its extensive test case production, `MeanDiff` only identifies one inconsistency each in `McSema`, `Revng`, and `RetDec`. None is triggered by interactions among instructions. Notably, LiftFuzz can also detect these inconsistencies. Contrarily, despite generating a mere 1/1000 of the test cases produced by `MeanDiff`, LiftFuzz obtained 26 unique inconsistencies. Interestingly, 11 of these inconsistencies are the results of interactions among instructions. These inconsistencies can lead to incorrect execution paths or program crashes, which could disrupt subsequent security analysis [32]. The result shows that simply expanding the diversity and volume of instructions won't necessarily boost test coverage or uncover more inconsistencies; it could only diminish efficiency. Besides, to our surprise, we have identified inconsistencies in frequently employed instructions. Next, we demonstrate several representative examples to offer a succinct introduction and conduct a thorough analysis.

> **RQ1 Answer:** LiftFuzz efficiently learns instruction interactions, generating high-quality and diverse test cases and outperforming `MeanDiff`. It uses only 1/1000 of `MeanDiff`'s test cases to identify 26 inconsistencies, a 767% increase compared to MeanDiff's findings.

5.2 RQ2: Inconsistencies Among Instructions

This section provides a case where LiftFuzz effectively shows its ability to generate test inputs that challenge the binary lifter and identify previously unknown inconsistencies. The case includes only a few straightforward and commonly used instructions.

Case 1: Inconsistency Among `jae` Instruction. The `jae` instruction performs a jump operation to the target instruction specified by the destination operand if the carry flag (`CF`) is zero. If the condition is not satisfied, the jump is not performed, and execution follows the instruction following the `jae` instruction [23]. The program counter increments to execute the next instruction. As shown in Listing 4, LiftFuzz generates several scenarios to verify whether the binary lifter can correctly translate `jae` instruction to IRs. Facing the same `jae` instruction respectively located in Line 4 and Line 6, `RetDec` translates it into two completely different IRs. The `jae` instruction, highlighted in yellow, located at Line 4, is translated into the IRs from Line 18 to Line 21, which is consistent with the actual instruction behavior. The `jae` instruction highlighted in red, located at Line 6, is translated into the IRs from Line 23 to Line 24. The function of these three lines is to return an undefined int64, which is inconsistent with the actual instruction behavior.

After further analysis, we find that `RetDec` identified the `jae` instruction at Line 4 and successfully translated the instruction into

[4] `Revng` does not have accurate version information. We downloaded it in July 2023.

```
1  /* Assembly codes */
2  0x4004fb:    mov      0x400505,  %rcx
3  0x400502:    00
4  0x400503:    jae      0x400505; <Block_1>
5  0x400505 <Block_1>:
6  0x400505:    jae      0x400513; <Block_2>
7   ...
8  0x400513 <Block_2>:
9  0x400513:    jae      0x400520; <Block_3>
10  ...
11 ------------------------------------------------
12 /* LLVM IR */
13 ; 0x4004fb
14 store volatile i64 4195579, i64* @_asm_program_counter
15 %27 = load i64, i64* inttoptr (i64 4195589 to i64*)
16 store i64 %27, i64* @rcx
17 ; 0x400503
18 store volatile i64 4195587, i64* _asm_program_counter
19 %28 = load i1, i1* cf
20 %29 = icmp eq i1
21 br i1 %29, label %dec_label_pc_400505, label %dec_label_pc_400505
22 ; 0x400505
23 dec_label_pc_400505:
24 ret i64 under
```

Listing 4: Case1: RetDec erroneously interprets Line 6 as data, leading to an inaccurate translation.

```
1  /* Assembly codes */
2  0x4004f4 <Block_0>:
3  0x4004f4:    neg      %eax
4  0x4004f6:    sbb      %eax,    %eax
5  0x4004f8:    movslq   %eax,    %rdx
6  ------------------------------------------------
7  /* LLVM IR */
8  ; 0x4004f6
9  store volatile i64 4195574, i64* @_asm_program_counter
10 %19 = load i64, i64* @rax
11 %20 = trunc i64 %19 to i32
12 %21 = load i64, i64* @rax
13 %22 = trunc i64 %21 to i32
14 %23 = load i1, i1* @cf
15 %24 = zext i1 %23 to i32
16 %25 = add i32 %22, %24
17 %27 = and i32 %20, 15
18 %28 = and i32 %25, 15
19 %29 = sub i32 %27, %28
20 %30 = load i1, i1* @cf
21 %31 = zext i1 %30 to i32
22 %32 = add i32 %29, %31
23 %33 = icmp ugt i32 %32, 15
24 store i1 %33, i1* @az
```

Listing 5: Case2: RetDec unfaithfully translates AF flag during the lifting of Line 4, sbb instruction.

the corresponding IRs. On the contrary, the disassembly result generated by RetDec indicates that RetDec recognized the instruction at Line 6 as data inside the code section. Hence, in this case, RetDec did not translate Line 6 as jae instruction.

Nonetheless, the issue has not been fully resolved. As shown at Line 21, RetDec utilizes a br instruction, redirecting control flow to two basic blocks [42]. Give that two parameters of the br instruction are both dec_label_pc_0x400505, the symbol used by RetDec to represent the IR branch of the instruction located at 0x400505, the address of the next instruction and the target instruction are the same, 0x400505. Consequently, whether the condition is to jump or not, 0x400505 will ultimately be executed. This implies that 0x400505 must be an instruction, not data. And the label dec_label_pc_0x400505 at Line 23 is the entry point to IRs of the instruction at address 0x400505. However, the translation results of Line 4 and Line 6 collectively suggest that RetDec possesses insufficient knowledge of disassembly to recognize that the address 0x400505 corresponds to an instruction rather than data. Thereby, RetDec has made a contradictory translation.

As mentioned in Section 5.1, all binary executables in our experiment contain debug information to ensure the accuracy of disassembly. Meanwhile, to eliminate the interference from the disassembler, we input the binary executable into Capstone [9], the disassembler used by RetDec, to test whether Capstone can successfully correctly disassemble all instructions. We observed that Capstone correctly disassembled all instructions of the test binary executable, including the jae instruction shown in Listing 4. This shows that the root cause of the problem is not the limitation of the underlying disassembler functionality but RetDec's failure to handle the interaction among instructions, leading to incorrectly translating an instruction into data and code simultaneously.

We also found similar situations with many other common instructions, which supports our insight that verifying the correctness of every single instruction in isolation is far away from the original intention and primary purpose of binary lifters. Complex interactions exist among instructions, preventing the binary lifter from correctly translating each instruction.

> **RQ2 Answer:** LiftFuzz discovered that the correct translation of a single instruction in one context does not necessarily guarantee its accurate translation in other different code scenarios.

5.3 RQ3: Case Studies on Inconsistencies Inside Instructions

This section provides two examples to illustrate how LiftFuzz utilizes its understanding of instruction interactions to validate whether the binary lifter can accurately utilize and propagate the effects within the instruction. Case 2 demonstrates how the inconsistency can lead to an instruction sending an incorrect control signal to the following instructions. Case 3 demonstrates how the inconsistency can result in an instruction's inability to handle the signal left by the previous instruction correctly.

Case 2: Inconsistency Inside sbb Instruction. The sbb instruction is to add the source operand and CF, and subtract the result from the destination operand. The OF, SF, ZF, AF, and CF are set according to the result [23]. As shown in Listing 5, LiftFuzz constructs a test case containing the sbb instruction and finds inconsistencies with the corresponding instruction in the emulation-style IRs generated by RetDec. To be precise, it is about calculating the AF. The AF (Auxiliary Carry) flag is a flag in x86 CPUs that indicates whether there was a carry-out or borrow into the least significant 4 bits during arithmetic operations [23].

Here is an overview of how RetDec calculates the AF flag in the IRs of the sbb instruction. RetDec uses az to represent AF. From Line 8 to Line 24, RetDec takes out the value of eax and the

value of eax+CF. Next, RetDec performs a subtraction operation on the lower four bits of eax and the lower four bits of eax+CF. Then, RetDec adds the value of CF to the result and performs an unsigned greater than operation with 15. Finally, RetDec writes the final result into AF. Since in Line 23, %32 and 15 perform unsigned greater than calculations, %33 will only become true when %32 is greater than 15 or less than 0. Therefore, RetDec will only set AF to 1 when the least significant 4 bits of eax is 0xf. However, in a real CPU, if CF is 1 after this instruction is executed, AF will be set to 1. Otherwise, AF will be 0. Hence, there is a huge inconsistency between the emulation-style IRs and the corresponding instruction.

The above is the case when the two operands are equal. When the first and second operands are different, the operation of IRs generated by RetDec is also inconsistent with the real instruction behavior. After we studied the source code of RetDec, we summarized this as a programming bug, since we found that there exist two programming problems. The first is that eax should intercept the lower four bits before adding it to CF, so that the result can be retained in int32 without losing. Another problem is that the operations on Line 20 to Line 22 are redundant. To sum it up, an operation is repeated in an error location.

Nevertheless, this does not solve the problem. After further research and conducting many tests on the physical CPU, we find that sbb instruction will set AF due to two situations. Remember that the sbb instruction performs addition of the source operand and CF, followed by subtraction of the result from the destination operand. There are two scenarios to consider: one arises from the carry generated after adding the lower four bits of eax to CF, and the other arises from the borrow obtained by subtracting eax+CF from eax. RetDec only considers the borrow situation and ignores the carry situation. Thereby, a simple modification of source code cannot address the issue and RetDec requires a re-implementation of the sbb instruction.

Case 3: Inconsistency Inside `imul` Instruction. The `imul` instruction is to perform a signed multiplication of two operands. As shown in Listing 6, LiftFuzz constructs a test case that includes an instruction at Line 6, `imul %rax, %rdx`, following an instruction at Line 4, `add %r11, %ecx`, and inputs it to McSema. The IRs generated by McSema for the instruction, `imul %rax, %rdx`, specifically lines 10 to 19, are responsible for setting the AF, ZF, PF, and SF flags. In this case, McSema uses the variables `@AF_2069_2ba8480` to represent AF, `@ZF_2071_2ba8480` to represent ZF, `@PF_2067_2ba8480` to represent PF, and `@SF_2073_2ba8480` to represent SF. In summary, the effect of these lines is to reset AF and ZF to zero, and adjust PF and SF based on the calculation results. Unfortunately, an inconsistency arises here. When the `imul` instruction is executed on a real CPU, it does not modify these four flags in the same manner but instead preserves the status set by the previous instruction `add %r11, %ecx` at Line 4.

The above two examples demonstrate that inconsistency causes the binary lifter to incorrectly set the `eflags` register, an important control and status register containing multiple flags that record the valuable information about the outcome of previous instructions and control the behavior of subsequent instructions. These flags are crucial for program execution, condition checking, and control flow. They allow the CPU to make decisions based on the outcome of previous operations, handle errors, and control program flow through conditional jumps and loops.

```
1  /* Assembly codes */
2  0x400504 <Block_1>:
3  ...
4  0x400508:    add      %r11d,  %ecx
5  0x40050b <Block_2>:
6  0x40050b:    imul     %rax,   %rdx
7  ...
8  ------------------------------------------------
9  /* LLVM IR */
10 %85 = call i32 @llvm.ctpop.i32(i32 ...)
11 %86 = trunc i32 %85 to i8
12 %87 = and i8 %86, 1
13 %88 = xor i8 %87, 1
14 store i8 %88, i8* @PF_2067_2ba8480
15 store i8 0, i8* @AF_2069_2ba8480
16 store i8 0, i8* @ZF_2071_2ba8480
17 %res_trunc.lobit.i.i69 = lshr i64 %retval.sroa.0.0.
       extract.trunc.i.i58, 63
18 %89 = trunc i64 %res_trunc.lobit.i.i69 to i8
19 store i8 %89, i8* @SF_2073_2ba8480
```

Listing 6: Case3: `McSema` inaccurately translates the `PF`, `AF`, `ZF` and `SF` flags in the lifting of Line 6,

RQ3 Answer: LiftFuzz discovered that inconsistencies within instructions can result in incorrect calculations and the propagation of effects among instructions.

5.4 RQ4: Root Causes of Inconsistencies

This section summarizes the inconsistencies found by LiftFuzz and the root causes of these inconsistencies. First, binary lifting is not merely a process of converting instructions from disassembly results into IRs on an instruction-by-instruction basis. Since a significant amount of high-level information, such as variable names, data types, and control structures, is lost during the compilation process, reconstructing the high-level structure and semantics of the original program poses a complex and formidable challenge. As demonstrated in Case 1, the binary lifter might encounter conflicting disassembly information, necessitating a crafted design to sift out incorrect interference information. Also, the binary lifter should filter in the correct information and employ IRs to restore and re-articulate the original structure of the program.

Second, instruction set manuals for modern processors are often incredibly detailed and extensive. As illustrated in Case 2, understanding and implementing support for every single instruction can be a daunting task. This is further complicated because different processors, even within the same family, may have different versions of instructions.

Third, the semantics of some instructions might not be clearly defined in the manual, or the manual might be ambiguous or have errors. As demonstrated in Case 3, this can lead to uncertainty about how to lift such instructions correctly. Furthermore, the behavior of some instructions can depend on the specific processor model, adding another layer of complexity.

RQ4 Answer: LiftFuzz discovered that the root causes are attributed to the lack of high-level information for program structure recovery. Additionally, the complexity of the modern instruction set manual and unclear definitions for certain semantics contribute to the challenges faced.

6 DISCUSSION

In this section, we will address certain limitations of our work and explore potential avenues for future research.

Alternative Assembly Language Model. Currently, we implement LiftFuzz with minGPT [26]. This reliance on minGPT restricts LiftFuzz's ability to fully comprehend the dataset and capture the intricate interactions among instructions. Consequently, LiftFuzz is unable to fully unleash its potential in generating highly efficient test cases. This limitation is acknowledged in our current work. However, despite the constraints imposed by hardware conditions and the model, LiftFuzz has successfully identified issues in three well-established binary lifters and has achieved commendable results. This outcome validates the correctness of our insights and the feasibility of our methodology. Hence, LiftFuzz's efficacy does not solely rely on large models, as even with smaller models, it can still yield favorable outcomes. In the future, we aim to explore the migration of LiftFuzz to the full-size GPT for testing purposes, and theoretically, the performance should be further improved.

Extend LiftFuzz to other lifters, IRs, ISAs. LiftFuzz currently centers around LLVM IR and x86-64. The test input generated by LiftFuzz can be directly applied to other binary lifters that utilize LLVM IR and operate on x86-64 because the test input is the binary executable, ensuring compatibility with any such binary lifters. Furthermore, the final verification process depends on the runtime information provided by the CPU, which is independent of the binary lifter itself. Meanwhile, expanding our work to encompass additional IRs and ISAs would not only enhance the dependability of binary lifters but also alleviate the developers' workload. For non-LLVM IRs, several instrumentations are needed to extract the runtime information if the lifted IR is recompilable; otherwise, an additional runtime module is required to make the lifted IR recompilable. For other ISAs, LiftFuzz adapts by learning from relevant datasets and incorporates hallucination for fuzzing input mutations. Then, LiftFuzz can generate corresponding test cases for validation. Consequently, LiftFuzz exhibits high adaptability. We believe that with minor modifications to the IR Integrator, LiftFuzz can be quickly employed by binary lifters utilizing alternative IRs.

Integrate LiftFuzz with `MeanDiff` `MeanDiff` can substitute randomly generated test instructions with those from LiftFuzz to validate the binary lifters. Nonetheless, `MeanDiff`'s limitation lies in detecting inconsistencies that are caused by inter-instruction interactions. This is due to `MeanDiff` processes and evaluates instructions individually, thereby missing out on the information shared between instructions.

7 RELATED WORKS

Past attempts to validate binary lifters can be generally divided into two main strategies: symbolic execution-based and testing-based.

7.1 Symbolic Execution-Based Approach

Dasgupta et al. [12] utilize formal semantics of LLVM and the x86 instruction to perform symbolic execution and differential analysis, emphasizing wide coverage. Additionally, they propose a program-level validation using a combination of validated instructions.

`MeanDiff` [27] propose N-version IR testing to validate three binary lifters: BAP [8], BINSEC [3], and PyVEX [46]. Their approach involves converting the different IRs generated by these three binary lifters for the same instruction into unified IR representations. These unified IR representations are then subjected to symbolic execution, which generates symbolic summaries used for differential analysis.

Reopt-vcg [21] is proposed to specifically verify Reopt [17]. Reopt-vcg takes annotations that establish the relationship between LLVM functions and addresses in the executable, generating proof obligations in the SMT-LIB. This work demands significant human resources and is prone to errors.

7.2 Testing-Based Approach

Chen et al. [10] enable the translation of ARM programs into x86 programs through a binary lifter. Then, they verify the accuracy of the binary lifter by executing both the original program and the translated program, and comparing the architectural states after each instruction. The evaluation of the validator is conducted using the ARM code compiled from EEMBC1.1 [16]. However, EEMBC1.1 was not designed to validate the binary lifter, resulting in the binary lifter not being efficiently verified. Unlike them, LiftFuzz can leverage interactions among instructions to generate specific test inputs that are utilized for testing the binary lifter.

Martignoni et al. [38] validates the "buggier and less complete" Lo-Fi emulator [5] by generating high-fidelity test inputs using symbolic execution. These test inputs are created based on the instruction semantics of a "faithful and more complete" Hi-Fi emulator [29]. They execute each test instruction twice, once on real hardware and then on the Lo-Fi emulator. By comparing the outputs of the Lo-Fi emulator with the outputs of the Hi-Fi emulator for these test inputs, they can identify and address discrepancies or bugs. Indeed, using the test input generated by the Hi-Fi emulator may not efficiently uncover potential problems in the Lo-Fi emulator. Therefore, while this approach provides some level of validation, it may not be as thorough or effective in identifying all possible issues. Earlier, Martignoni et al. [36, 37] propose hardware-cosimulation based testing on QEMU [5] and Bochs [29]. They compared the machine state between the physical CPU and the emulator after executing randomly generated test inputs to discover inconsistencies. Due to the random generation of all test instructions, the efficiency of generating test instructions and triggering binary lifter problems is significantly constrained.

Our work. Prior works all assumed that the binary lifter processes instructions without taking contexts into account, consistently generating identical IR for each identical instruction. This results in each instruction being validated in isolation, overlooking the potential interactions among instructions. And we have discovered that this assumption is incorrect. According to our observation, the binary lifter processes the same instruction differently due to different contexts. To fill this gap, LiftFuzz first learns the interactions

among instructions and generates test inputs to verify whether the binary lifter can handle complex interactions among instructions.

8 CONCLUSION

In this paper, we introduce LiftFuzz, a novel framework that leverages instruction context-aware fuzzing to validate binary lifters. Contrary to existing validation methods that predominantly focus on isolated instructions, neglecting the interactions among instructions, LiftFuzz employs an assembly language model to comprehend and learn from the interactions among instructions, thereby generating test cases with that knowledge. We evaluate LiftFuzz with against three predominant binary lifters, `McSema`, `Revng`, and `RetDec`. In total, LiftFuzz discovers 26 inconsistencies, including a previously uncovered category.

9 ACKNOWLEDGEMENT

We want to thank our anonymous reviewers for their valuable comments. This work was supported in part by National Key Research & Development Project of China (Grant No. 2019YFB1804400), Hong Kong S.A.R. Research Grants Council (RGC) General Research Fund No. 14209720, and research fund (TA2217345) from Alipay (Hangzhou) Information Technology Co., Ltd. Jiongyi Chen was supported by the Natural Science Foundation of China (Grant No. 62302508) and Research Funding of NUDT (Grant No. ZK22-53).

REFERENCES

[1] Avast Threat Labs. Accessed: September 2023. Avast. https://github.com/avast/retdec.
[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR* 1409.0473 (2014).
[3] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, et al. 2011. The BINCOA Framework for Binary Code Analysis. In *CAV*. 165–170.
[4] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *NDSS'2018*.
[5] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX*, Vol. 41. 10–5555.
[6] Derek Bruening. 2004. *Efficient, transparent, and comprehensive runtime code manipulation.* Ph. D. Dissertation. Massachusetts Institute of Technology, USA.
[7] Derek Bruening, Timothy Garnett, and Saman P. Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *CGO*. IEEE, 265–275.
[8] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *CAV*. Springer, 463–469.
[9] Capstone. Accessed: October 2023. http://www.capstone-engine.org/.
[10] Jiunn-Yeu Chen, Wuu Yang, Bor-Yeh Shen, et al. 2015. Automatic validation for binary translation. *Computer Languages, Systems & Structures* 43 (2015), 96–115.
[11] Cristina Cifuentes and Mike Van Emmerik. 2000. UQBT: Adaptive Binary Translation at Low Cost. *Computer* 33, 3 (2000), 60–66.
[12] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S. Adve, et al. 2020. Scalable validation of binary lifters. In *PLDI*. 655–671.
[13] Sandeep Dasgupta, Daejun Park, et al. 2019. A complete formal semantics of x86-64 user-level instruction set architecture. In *PLDI*. 1133–1148.
[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).
[15] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *SP*. IEEE, 472–489.
[16] EEMBC. Accessed: October 2023. EEMBC. https://www.eembc.org/techlit/.
[17] GaloisInc. Accessed: October 2023. Reopt. https://github.com/GaloisInc/reopt.
[18] GNU. Accessed: October 2023. The GNU C Library (glibc). https://www.gnu.org/software/libc/documentation.html.
[19] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*, Vol. 8. 151–166.
[20] Google Could. Accessed: April 2024. Evaluating models. https://cloud.google.com/translate/automl/docs/evaluate.
[21] Joe Hendrix, Guannan Wei, and Simon Winwood. 2019. Towards verified binary raising. In *Workshop on Instruction Set Architecture Specification 2019*, Vol. 6.
[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[23] Intel. Accessed: September 2023. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.
[24] Frederick Jelinek, Robert L. Mercer, Lalit R. Bahl, and Janet M. Baker. 1977. Perplexity—a measure of the difficulty of speech recognition tasks. *Journal of the Acoustical Society of America* 62 (1977).
[25] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. SymLM: Predicting Function Names in Stripped Binaries via Context-Sensitive Execution-Aware Code Embeddings. In *CCS*. 1631–1645.
[26] Andrej Karpathy. Accessed: October 2023. minGPT. https://github.com/karpathy/minGPT/.
[27] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungil Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing intermediate representations for binary analysis. In *ASE*. 353–364.
[28] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, et al. 2010. PEBIL: Efficient static binary instrumentation for Linux. In *ISPASS*. IEEE, 175–183.
[29] Kevin P Lawton. 1996. Bochs: A portable pc emulator for unix/x. *Linux Journal* 1996, 29es (1996), 7–es.
[30] Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and Bill Dolan. 2015. A diversity-promoting objective function for neural conversation models. *arXiv preprint arXiv:1510.03055* (2015).
[31] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. Palmtree: Learning an assembly language model for instruction embedding. In *CCS*. 3236–3251.
[32] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. 2022. Sok: Demystifying binary lifters through the lens of downstream applications. In *SP*. 1100–1119.
[33] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *ICLR*. https://openreview.net/forum?id=Bkg6RiCqY7
[34] Chi-Keung Luk, Robert S. Cohn, Robert Muth, et al. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*. 190–200.
[35] Ben Mann, N Ryder, M Subbiah, J Kaplan, P Dhariwal, A Neelakantan, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).
[36] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. Testing system virtual machines. In *ISSTA*. 171–182.
[37] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU emulators. In *ISSTA*. 261–272.
[38] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, et al. 2010. N-version disassembly: differential testing of x86 disassemblers. In *ISSTA*. 265–274.
[39] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *ACL*. 311–318.
[40] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, et al. 2021. Stateformer: Fine-grained type recovery from binaries using generative state modeling. In *ESEC/FSE'21*. 690–702.
[41] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680* (2020).
[42] LLVM Project. Accessed: September 2023. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.
[43] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. *OpenAI* (2018).
[44] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* (2019).
[45] rev.ng. Accessed: September 2023. Revng. https://github.com/revng/revng.
[46] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS'22nd*.
[47] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *NeurIPS*. 3104–3112.
[48] Guy Tevet and Jonathan Berant. 2020. Evaluating the evaluation of diversity in natural language generation. *arXiv preprint arXiv:2004.02990* (2020).
[49] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* 27, 8 (1984), 761–763. https://doi.org/10.1145/358198.358210
[50] Trail of Bits research team. Accessed: September 2023. Mcsema. https://github.com/lifting-bits/mcsema.
[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 5998–6008.
[52] Vector 35. Accessed: May 2024. Binary Ninja. https://binary.ninja/.
[53] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable disassembling. In *USENIX*. 627–642.
[54] Tao Wang, Jun Liu, Xiaoning Zhang, Kehuan Zhang, et al. 2016. XcodeGhost: A Large-Scale Apple App Store Malware. *IEEE Access* 4 (2016), 5183–5191.
[55] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. 283–294.
[56] Xinyuan Zhang, Yi Yang, Siyang Yuan, et al. 2019. Syntax-infused variational autoencoder for text generation. *arXiv preprint arXiv:1906.02181* (2019).
[57] Yaoming Zhu, Sidi Lu, Lei Zheng, Jiaxian Guo, Weinan Zhang, et al. 2018. Texygen: A benchmarking platform for text generation models. In *SIGIR*. 1097–1100.