EXAMINER: automatically locating inconsistent instructions between real devices and CPU emulators for ARM

MUHUI JIANG, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, Hong Kong

TIANYI XU, Zhejiang University, Hangzhou, Zhejiang, China

YAJIN ZHOU, Zhejiang University, Hangzhou, Zhejiang, China

YUFENG HU, Zhejiang University, Hangzhou, Zhejiang, China

MING ZHONG, Zhejiang University, Hangzhou, Zhejiang, China

LEI WU, Zhejiang University, Hangzhou, Zhejiang, China

View all

Published: 22 February 2022

Citation in BibTeX format

ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems
February 28 - March 4, 2022
Lausanne, Switzerland

Conference Sponsors:
SIGARCH
SIGPLAN
SIGOPS

EXAMINER: Automatically Locating Inconsistent Instructions between Real Devices and CPU Emulators for ARM

Muhui Jiang[*]
The Hong Kong Polytechnic University;
Zhejiang University, China
csmjiang@comp.polyu.edu.hk

Tianyi Xu
Zhejiang University, China
xtyi@zju.edu.cn

Yajin Zhou[†]
Zhejiang University, China
yajin_zhou@zju.edu.cn

Yufeng Hu
Zhejiang University, China
yufenghu@zju.edu.cn

Ming Zhong
Zhejiang University, China
tracym@zju.edu.cn

Lei Wu
Zhejiang University, China
lei_wu@zju.edu.cn

Xiapu Luo
The Hong Kong Polytechnic University,
China
csxluo@comp.polyu.edu.hk

Kui Ren
Zhejiang University, China
kuiren@zju.edu.cn

ABSTRACT

Emulators are widely used to build dynamic analysis frameworks due to its fine-grained tracing capability, full system monitoring functionality, and scalability of running on different operating systems and architectures. However, whether emulators are consistent with real devices is unknown. To understand this problem, we aim to automatically locate inconsistent instructions, which behave differently between emulators and real devices.

We target the ARM architecture, which provides machine-readable specifications. Based on the specification, we propose a sufficient test case generator by designing and implementing the first symbolic execution engine for the ARM architecture specification language (ASL). We generate 2,774,649 representative instruction streams and conduct differential testing between four ARM real devices in different architecture versions (i.e., ARMv5, ARMv6, ARMv7, and ARMv8) and three state-of-the-art emulators (i.e., QEMU, Unicorn, and Angr). We locate a huge number of inconsistent instruction streams (171,858 for QEMU, 223,264 for unicorn, and 120,169 for Angr). We find that undefined implementation in ARM manual and bugs of emulators are the major causes of inconsistencies. Furthermore, we discover 12 bugs, which influence commonly used instructions (e.g., `BLX`). With the inconsistent instructions, we build three security applications and demonstrate the capability of these instructions on detecting emulators, anti-emulation, and anti-fuzzing.

[*]Most of the work was completed when the first author was visiting Zhejiang University.
[†]Corresponding author.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507736

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**.

KEYWORDS

Emulator, Differential Testing, Inconsistent Instructions

ACM Reference Format:
Muhui Jiang, Tianyi Xu, Yajin Zhou, Yufeng Hu, Ming Zhong, Lei Wu, Xiapu Luo, and Kui Ren. 2022. EXAMINER: Automatically Locating Inconsistent Instructions between Real Devices and CPU Emulators for ARM. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507736

1 INTRODUCTION

A CPU emulator is a powerful tool as it provides fundamental functionalities (e.g., tracing, record and replay) for the dynamic analysis. Though hardware-based tracing techniques exist, they have limitations compared with software emulation. For example, ARM ETM has a limited Embedded Trace Buffer (ETB). The size of ETB of the Juno Development Board is 64KB [1] [1]. On the contrary, software emulation is capable of tracing the whole program, provides user-friendly APIs for runtime instrumentation, and is supported by multiple operating systems and architectures. Nevertheless, software emulation complements the hardware-based tracing techniques and provides rich functionalities for dynamic analysis frameworks.

Indeed, many dynamic analysis frameworks [24, 26–29, 31, 32, 34, 36, 37, 40, 41, 45, 50, 61, 66] are built based on the state-of-the-art CPU emulators (e.g., QEMU [12], Unicorn [17], Angr [3]) to conduct malware analysis, live-patching, crash analysis and etc. Meanwhile, many fuzzing tools utilize CPU emulators to fuzz binaries, e.g., the QEMU mode of AFL [2], Unicornfuzz [51], FirmAFL [67], P2IM [33], HALucinator [30], and TriforceAFL [15].

The wide adoption of software emulation usually has an implicit assumption that the execution result of an instruction on

[1]The ETB size of different SoCs may be different. However, it is usually limited due to the chip cost and size.

the CPU emulator and the real device is identical, thus running a program on the CPU emulator can reflect the result on the real hardware. *However, whether this assumption really holds in reality is unknown.* In fact, the execution result can be different (as shown in our work), either because the CPU emulator has bugs or it uses a different implementation strategy from the real device. These differences impede the reliability of emulator-based dynamic analysis. For instance, the malware can abuse the differences to protect the malicious behaviors from being analyzed in the emulator [38, 39, 49, 58].

In this work, we aim to automatically locate inconsistent instructions, which behave differently between emulators and real devices, for the ARM architecture. In this paper, *instruction* denotes the category in terms of functionality, which is usually represented by its name in ARM manual. For example, `STR (immediate)` is an instruction, which aims to store a word into memory. Automatically locating inconsistent instructions is not easy. *First*, ARM architecture has multiple versions (e.g., ARMv5, ARMv6, ARMv7, and ARMv8), different register widths (16 bits or 32 bits) and instruction sets. Besides, it has mixed instruction modes (ARM, Thumb-1, and Thumb-2). Thus, how to generate sufficient *instruction streams*, which denotes the bytecode of an instruction, to cover these variants, while at the same time generating only necessary ones to save the time cost, is the first challenge. Note that if we naively enumerate 32-bit instruction streams, the number of test cases would be 2^{32}, which is inefficient to be evaluated. Meanwhile, randomly generated instruction streams are not representative and many instructions are not covered (Section 4.1). *Second*, for each test case, we should provide a deterministic environment to execute the single instruction stream and automatically compare the result after the execution. This requires us to set up the same CPU state before the execution and compare the state afterwards.

Previous works [52–55], which target x86/x64 architectures, provide valuable insights. However, they either use randomized test cases or rely on the emulator or hardware to generate the test cases, which is not sufficient and the results may be biased. Meanwhile, existing designed differential testing frameworks (e.g., EmuFuzzer [55]) require that the emulator should be running inside the compared real device, which are not scalable. Furthermore, whether the findings can be applied on the ARM architecture is unknown. Though recent work (i.e., iDEV [57]) studies the semantic deviation issue of ARM instructions, they lack a systematic way to generate sufficient test cases. Instead, they enumerate a huge number of (i.e., 33 million) redundant test instructions that cannot cover all the instruction behaviors. Meanwhile, they only focus on the triggered signals during the execution process without checking the whole CPU state, resulting in many inconsistent instructions unexplored. Furthermore, the evaluation is limited to ARMv7 and QEMU. There are many other ARM architectures (e.g., ARMv5, ARMv6, and ARMv8) and lightweight but also popular emulators (i.e., Unicorn, Angr), which many frameworks are based on [25, 37, 51, 64]. We will discuss the major differences between iDEV and our work in Section 5.

Our system is able to automatically locate inconsistent instructions in a systematic mechanism with the following two key insights, which have been neglected by existing works.

Syntax and semantics aware test case generator To generate representative test cases, we propose a syntax and semantics aware methodology. Each ARM instruction consists of several *instruction encodings*, which describe which parts of the instruction are constant and which parts can be mutated (Fig. 1a). The non-constant parts are called *encoding symbols*. Each instruction encoding has specific decoding and execution logic, which is is expressed in the ARM's Architecture Specific Language (ASL) [59] . We call it *ASL code* (Fig. 1b and 1c). ASL code executes based on the concrete values of the encoding symbols. In this case, we first take the syntax-aware strategy. For each encoding symbol, we mutate it based on pre-defined rules. For instance, for the immediate value symbol, the values in the mutation set cover the maximum value, the minimum value and a fixed number of random values. This strategy generates syntactically correct instructions. We further take a semantics-aware strategy to generate more test cases as the previous strategy may only cover limited instruction semantics (Section 2.2). To this end, we extract the constraints, which influence the execution path, in ASL code. We solve the constraints and their negations by designing and implementing *the first symbolic execution engine for ASL* to find the satisfied values of the encoding symbols. By doing so, the generated test cases can cover different semantics of an instruction.

Deterministic differential testing engine Comparing the execution result of emulators/real devices with the ARM specification directly relies on a precise ASL interpreter. However, the precision of the ASL interpreter cannot be guaranteed. In this case, we propose a differential testing engine, which uses the generated test cases as inputs. To get a deterministic testing result, we provide the same context when executing an instruction stream on a real CPU and an emulator. We complete this goal by inserting the prologue and epilogue instructions. The prologue instructions aim to set the execution environment while the epilogue instructions will dump the execution result for comparison to check whether the tested instruction stream is an inconsistent one.

We implement a prototype system called Examiner, which consists of a test case generator and a differential testing engine. Our test case generator generated 2, 774, 649 instruction streams that cover all the 1, 998 ARM instruction encodings in four instruction sets (i.e., A64, A32, T32, and T16). On the contrary, the same number of randomly generated instruction streams can only cover 54.5% instructions encodings, which shows the sufficiency of our test case generator. We then feed these test cases into our differential testing engine. By comparing the result between the state-of-the-art emulator (i.e., QEMU [12]) and real devices in different architectures (ARMv5, ARMv6, ARMv7, and ARMv8), our system detected 171, 858 inconsistent instruction streams, which cover 26.6% of the instruction encodings. To demonstrate the generalization of Examiner, we further apply Examiner on two more CPU emulators (i.e., Unicorn [17] and Angr [3]) and 223, 264 and 120, 169 inconsistent instructions are located, respectively. We then explore the root causes. It turns out that implementation bugs and the undefined implementation in the ARM manual are the major causes. We discovered 12 bugs (4 in QEMU [8, 13, 16, 22], 3 in Unicorn [9], 5 in Angr [7, 18–21]) and all of them have been confirmed by developers. These bugs can influence commonly used instructions (e.g., `BLX`) and can even crash the emulators (e.g., QEMU and Angr).

To show the usefulness of our findings, we further build three applications, i.e., emulator detection, anti-emulation, and anti-fuzzing. By (ab)using inconsistent instructions, a program can successfully detect the existence of the CPU emulator and prevent the malicious behavior from being monitored. Besides, the coverage of the program being fuzzed inside an emulator can be highly decreased. Note that we only use these applications to demonstrate the usage scenarios of our findings. There may exist other applications. In summary, our work makes the following main contributions.

Sufficient test case generator We propose a test case generator by introducing the first symbolic execution engine for ARM ASL code. It can generate representative instruction streams that sufficiently cover different instructions (encodings) and their semantics.

Effective prototype system We implement a prototype system named Examiner that consists of a test case generator and a differential testing engine. Our experiments showed Examiner is general and can automatically locate inconsistent instructions.

New findings We explore and report the root cause of the inconsistent instructions. Implementation bugs of emulators and undefined implementation in ARM manual are the major causes. Furthermore, 12 bugs have been discovered and confirmed by the developers. Some of them influence commonly used instructions (e.g., `BLX`) and can make the emulators crash.

To engage with the community, we release the source code of our system in https://github.com/valour01/examiner.

2 BACKGROUND AND MOTIVATION

2.1 CPU Emulators

CPU emulators usually support multiple CPU architectures. When executing an instruction stream, the emulator first decodes the instruction stream and converts it into intermediate representations (IR). After generating the IR, emulators like QEMU will further translate the IR into host machine instructions, which will be executed on the host machine directly. As the host machine provides an operating system, other features like syscalls and the POSIX signals are also supported. Based on QEMU, Unicorn, which provides friendly APIs to build different tools, is proposed. Unicorn aims to emulate the CPU operations only and remove the other supports (e.g., signals) to keep it as a lightweight one. Other binary frameworks like Angr also support CPU emulation. Users can specify the entry address and execute the target instruction step by step in Angr.

2.2 Motivation

Examiner can be used to find inconsistent instructions, which can be used in many scenarios (Section 4.4), automatically. We illustrate how Examiner can detect the inconsistent instruction and find the bugs of emulator with a motivation example.

2.2.1 The Encoding Schema and Semantics. Fig. 1 shows one of the encoding schemas of instruction `STR (immediate)` and the corresponding ASL code for decoding and execution logic. According to the encoding schema in Fig. 1a, the value is constant (i.e., `111110000100` and `1`) for offset [31:20] and [7:7]. The values in other offsets include 6 encoding symbols and they are `Rn`, `Rt`, `P`, `U`, `W`, and `Imm8`.

31	30	29	28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11	10	9	8	7 6 5 4 3 2 1 0
1	1	1	1	1	0	0	0	0	1	0	0	Rn	Rt	1	P	U	W	Imm8

(a) The encoding schema of the STR (immediate) instruction in Thumb-2.

```
1   if Rn == '1111' || (P == '0' && W == '0') then
        UNDEFINED;
2   t = UInt(Rt);
3   n = UInt(Rn);
4   imm32 = ZeroExtend(imm8, 32);
5   index = (P == '1');
6   add = (U == '1');
7   wback = (W == '1');
8   if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

(b) The ASL code for decoding the instruction.

```
1   offset_addr = if add then (R[n] + imm32) else (R[n]
        - imm32);
2   address = if index then offset_addr else R[n];
3   MemU[address,4] = R[t];
4   if wback then R[n] = offset_addr;
```

(c) The ASL code for executing the instruction.

Figure 1: A motivation example.

Fig. 1b shows the decoding ASL code. Note that the ASL code is simplified for presentation. The complete code can be found on the official ARM site [4].

- In Line 1, the symbol `Rn`, `P`, and `W` will be checked. If the condition is satisfied, the instruction stream will be treated as an `UNDEFINED` one. Consequently, a `SIGILL` signal or undefined instruction exception will be raised by emulators.
- In line 2 and 3, the symbol `Rt` and `Rn` will be converted to unsigned integer `t` and `n`, respectively. Similarly, the symbol `imm8` will be extended into a 32-bit integer `imm32`. In line 5, 6, and 7, symbol `index`, `add`, and `wback` will be assigned according to the value of `P`, `U`, and `W`, respectively.
- In line 8, the symbol `t`, `wback`, and `n` will be checked. If the condition is satisfied, the instruction stream should be treated as `UNPREDICTABLE`. According to ARM's manual, the behavior of an `UNPREDICTABLE` instruction stream is not defined. The processor vendors and the emulator developers can choose an implementation that they think is proper.

Similarly, Fig. 1c shows the ASL code for the execution logic of the instruction. The ASL code in Fig. 1b and Fig. 1c defines the semantics of the instruction.

2.2.2 Test Case Generation. By analyzing the encoding schema, Examiner generates the test cases by mutating the non-constant fields, including `Rn`, `Rt`, `P`, `U`, `W` and `Imm8`. This can generate syntactically correct instructions. However, this step is not enough, since it may not generate the values that satisfy the condition in the ASL code. For instance, one constraint in line 8 of Fig. 1b is `t == 15`. The random values generated in the first step may not satisfy this expression (`Rt` is not equal to 15). To this end, we leverage a constraint solver to find the concrete value of `Rt` that satisfies the constraint, i.e., `15`. We take similar actions to solve the constraints for other symbols in line 1 (`add`), 2 (`index`) and 4 (`wback`) of Fig. 1c. To cover different execution paths of ASL code, we will also solve

```
1    static bool op_store_ri(DisasContext *s,
         arg_ldst_ri *a, MemOp mop, int mem_idx)
2    {
3        ISSInfo issinfo = make_issinfo(s, a->rt, a->p,
             a->w) | ISSIsWrite;
4        TCGv_i32 addr, tmp;
5
6        // Rn=1111 is UNDEFINED for Thumb;
7
8    +   if (s->thumb && a->rn == 15) {
9    +       return false;
10   +   }
11
12       addr = op_addr_ri_pre(s, a);
13
14       /*omitted QEMU code*/
15
16       return true;
17   }
```

Figure 2: Original code of QEMU and the patch for function op_store_ri, which aims to translate STR instruction

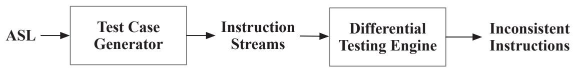


Figure 3: The work flow of our system

the negations of the constraints. During this process, we generated 576 instruction streams as test cases in total.

2.2.3 Differential Testing. We feed each instruction stream into our differential testing engine (Section 3.2), which adds prologue and epilogue instructions. The prologue instructions first set the initial execution context before executing the instruction stream. After the instruction stream is executed, the epilogue instructions will dump the result for comparison. We then execute these instructions on both emulators and real devices (e.g., RasberryPi 2B). By comparing the execution result, we confirm that `0xf84f0ddd` is an inconsistent instruction stream. Specifically, It will generate a `SIGILL` signal in a real device while a `SIGSEGV` signal in QEMU. We further analyzed the root cause and successfully disclosed a bug in QEMU [13]. According to Fig. 1a, the concrete value of `Rn` of the instruction stream `0xf84f0ddd` is `1111`. As shown in the ASL code (line 1) in Fig. 1b. it is an `UNDEFINED` instruction stream. However, QEMU does not properly check this condition. Fig. 2 shows the (patched) function (i.e., `op_store_ri`) in QEMU for decoding the instruction `STR` (immediate). It continues the decoding process directly from line 12 without any check. We then submit this bug to QEMU developers and the patch is issued (as shown in line 8-10).

3 DESIGN AND IMPLEMENTATION

Figure 3 shows the workflow of Examiner, which consists of a test case generator and a differential testing engine. First, the test case generator retrieves the ASL code to generate the test cases (Section 3.1). Then, the differential testing engine receives the generated test cases and conducts differential testing between the emulators and real devices (Section 3.2). The instructions leading to different execution results are identified as inconsistent instructions. We further analyze the identified inconsistent instructions to understand the root cause of them and how they can be (ab)used.

Algorithm 1: The algorithm to generate test cases.

```
   Input: The encoding diagram: $I\_Encode$;
   The decoding ASL code: $I\_Decode$;
   The execution ASL code: $I\_Execute$
   Output: The generated test cases: $T$;
1  Function Generate($I\_Encode$,$I\_Decode$,$I\_Execute$):
2      $Symbols, Constants, Constraints$ = ParseASL($I\_Encode$,
         $I\_Decode, I\_Execute$)
3      for $S$ in $Symbols$ do
4          $S.MutationSet$ = InitSet($S$)
5      for $C$ in $Constants$ do
6          $C.MutationSet$ = [ConstantValue]
7      for $C$ in $Constraints + NegatedConstraints$ do
8          ValueSet = SolveConstraint($C, Symbols, I\_Decode, I\_Execute$)
9          for $V, S$ in $ValueSet$ do
10             if $V$ not in $S.MutationSet$ then
11                 $S.MutationSet$ add $V$
12     $MutationSets = [S.MutationSet + C.MutationSet]$
13     $TestCase$ = CartesianProduct($MutationSets$)
14     return $T$
```

3.1 Test Case Generator

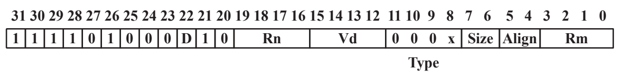
In theory, for a 32-bit instruction, there exist $2^{32} = 4,294,967,296$ possible instruction streams, which is too large for exhaustive exploration. In our work, we need to generate representative test cases that cover most behaviors of an instruction.

Specifically, we first parse the encoding schema to retrieve the encoding symbols and then infer the type for symbols, e.g., a register index or an immediate value. After that, we generate an initialized mutation set with pre-defined rules, which are shown in Table 1, for each type of symbol. For instance, we generate the maximum, minimum and random values for an immediate value. Then, we develop a symbolic execution engine to solve the constraints in the ASL code for the decoding and execution logic. This step can add more values to the mutation set to satisfy the constraints in the ASL code. At last, we generate instruction streams based on the values of encoding symbols.

Algorithm 1 shows how we generate the test cases. For each instruction, ARM provides an XML file to describe the instruction. We extract the encoding schemas and the corresponding ASL code for decoding and execution by parsing the XML file. We first retrieve the encoding symbols (*Symbols*) and constant values (*Constants*) in the encoding schema, as well as *Constraints* in ASL code (line 2). We then iterate over the *Symbols* and generate the *MutationSet* for each symbol (line 3-4), which will be introduced in detail in Section 3.1.1. Note this is the initial mutation set for each symbol. For the *Constants*, the *MutationSet* contains only the fixed value (line 5-6). After that, we solve the constraints and their negations to generate a new mutation set (i.e., *ValueSet*) for each symbol (line 7-8), which will be introduced in detail in Section 3.1.2. Then we check whether the solved value for each symbol is in the symbol's *MutationSet* (line 9). If not, we append it to the symbols' *MutationSet* (line 10-11). After that, we combine the *MutationSet* of both symbols and constants to get the *MutationSets* (line 12). Finally, considering all the possible combinations of the candidates in the *MutationSet* for each symbol, we conduct the Cartesian Product on the *MutationSets* to get the test cases (line 13).

Table 1: The rules of initializing the mutation set.

Type of Symbol Name	Mutation Set
Register Index	0 (R0); 1 (R1); 15 (PC); Random index values
Immediate Value in N bits	Maximum value: 2^N -1; Minimum value: 0; (N-2) Random Value from the enumerated values
Condition	"1110" (Always execute)
Others in 1 bit	"0"; "1"
Others in N bit (N >1)	N random value from the enumerated values



(a) Encoding diagram of instruction VLD4 in A32 instruction set

```
1   case type of
2       when '0000'
3           inc = 1;
4       when '0001'
5           inc = 2;
6   if size == '11' then UNDEFINED;
7   alignment = if align == '00' then 1 else 4 << UInt(
        align);
8   ebytes = 1 << UInt(size);
9   elements = 8 DIV ebytes;
10  d = UInt(D:Vd);
11  d2 = d + inc;
12  d3 = d2 + inc;
13  d4 = d3 + inc;
14  n = UInt(Rn);
15  m = UInt(Rm);
16  wback = (m != 15);
17  register_index = (m != 15 && m != 13);
18  if n == 15 || d4 > 31 then UNPREDICTABLE;
```

(b) Decoding code of instruction VLD4 in A32 instruction set

Figure 4: Test case generator example.

3.1.1 Initialize Mutation Set. In the phase of initializing the mutation set, we consider the types of different symbols and aim to cover different values according to their types. In particular, we infer the type based on the symbol name. For instance, a symbol that represents a register index usually has the name `Rd`, `Rm`, `Rn`, etc. As for the immediate value, the symbol name is usually `immn` where `n` represents the length of the value. For example, the symbol `imm8` represents an 8-bit immediate value.

Table 1 shows the rules to initialize the mutation set. For a register index, we include the `PC` register (index 15), `R0`, `R1`, and random values in the set. The register `R0` and `R1` are used to represent the return value for function calls. As for the PC, it can explicitly change the execution flow of the program. Thus, the register index in many instruction encodings cannot be 15. We include it in the mutation set to cover such cases. For the immediate value, the maximum and minimum value are the two boundary values that need to be covered. Apart from this, we randomly select (N-2) values, where N represents the bit length of the symbol. Note that enumerating all the values for one symbol is not realistic because immediate values may have up to 24 bits, resulting in 2^{24} = 16777216 candidates.

3.1.2 Solve Constraints. The execution paths of the ASL code depend on whether the constraints are met or not, which is decided by the value of encoding symbols. To make our test case representative, the generated test cases should cover as many paths as possible. To this end, we design and implement a symbolic execution engine for the ASL code. Specifically, we assign symbolic values for encoding symbols. Then we generate the symbolic expressions according to the ASL code. After that, we retrieve the constraints including the symbolic expression and feed them to SMT solvers. In this case, we can find the concrete values of the encoding symbols that satisfy or not satisfy the constraint.

Figure 4 shows a concrete example. In line 18, there is a symbolic expression `d4` and a constraint $d4 > 31$. All the related statements (line 3, 5, 10, 11, 12, and 13) are retrieved via backward slicing and highlighted in the green color. To solve this constraint, we conduct backward symbolic execution. Specifically, the symbol `d4` is calculated by the expression $d4 = d3 + inc$ in line 13. Thus, the constraint is converted to $d3 + inc > 31$. Given the relationship between `d3` and `d2` in line 12, and between `d2` and `d1` in line 11, we further convert it to $UInt(D : Vd) + 3 \times inc > 31$. The expression `UInt(D:Vd)` is converted to $Vd + 2^4 \times D$ as the symbol `Vd` has 4 bits. Thus, we have the constraint $Vd + 16 \times D + 3 \times inc > 31$. Symbol inc is assigned at line 3 or line 5. Thus, the constraint is $inc == 1$ *or* $inc == 2$. Apart from this, we need to consider the length of each symbol. Since D is one bit and Vd has four bits. Their constraints are $D \geq 0$ *and* $D < 2$, $Vd \geq 0$ *and* $Vd < 16$.

We feed all these constraints to the SMT solver. It returns one solution that Vd is 13, D is 1, and inc is 2. We then negate the constraint $d4 > 31$ and repeat the above-mentioned process. In this case, the solution is Vd is 0, D is 0, and inc is 1. Thus, the generated $ValueSet$ contains three symbols and each symbol has two candidate values. Note inc's value depends on $Type$'s value. As we will also solve the constraint $Type$ == '0000' and $Type$ == '0001', the final mutation set of $Type$ must contain the value that can make inc to be either 1 or 2. Due to the Cartesian Product between each symbol's mutation set, we can always generate the instruction streams that can satisfy the constraint $d4 > 31$ and its negation.

Note that the path explosion in symbolic execution is not an issue since the decoding and execution ASL code has limited constraints, resulting in limited paths. Meanwhile, we model the utility functions (e.g., UInt) so that the symbol will not be propagated into these functions. Our experiment in Section 4.1 shows that we can generate all the test cases within 4 minutes.

3.2 Differential Testing Engine

3.2.1 Model the CPU. The differential testing engine receives the generated instruction streams, and detects inconsistent ones. Formally, given one instruction stream I, we denote the state before the execution of I as the initial state CPU_I and the state after the execution of I as the final state CPU_F. We denote the CPU T's initial state $CPU_I(T)$ with the tuple $< PC_T, Reg_T, Mem_T, Sta_T >$. PC denotes the program counter, which points to the next instruction that will be executed. Reg denotes the registers used by processors while Mem denotes the memory space that the tested instruction I may write into. Note we do not consider the whole memory space as comparing the whole memory space is time- and resource-consuming. Sta denotes the status register, which is $APSR$ in ARM architecture. We denote the CPU T's final state $CPU_F(T)$ with the tuple $[PC_T, Reg_T, Mem_T, Sta_T, Sig_T]$. Inside $CPU_F(T)$, all the other attributes have the same meanings as they are inside $CPU_I(T)$ except Sig. Sig denotes the signal or exception that the instruction

Table 2: The statistics of the generated instruction streams. "EXAMINER " denotes the number of generated test cases by our test case generator. "Random" denotes the number of randomly generated test cases. "Ratio" denotes the percentage of dividing "Random" by "EXAMINER ". Note that one instruction may have different instruction encodings for different instruction sets. The total number of instructions for A32, T32, and T16 is 489.

Instruction Set	Time (s) of	Instruction Stream			Instruction Encoding			Instruction			Covered Constraints		
	EXAMINER	EXAMINER	Random	Ratio	EXAMINER	Random	ratio	EXAMINER	Random	ratio	EXAMINER	Random	Ratio
A64	70.51	1,094,700	421,645	38.5%	839	265	31.6%	581	178	30.6%	3,436	934	27.2%
A32	75.05	870,221	578,845	66.5%	550	415	75.5%	481	361	75.1%	4,718	3,725	79%
T32	74.58	808,770	34,598	4.2%	531	351	66.1%	451	283	62.7%	4,425	3,203	72.3%
T16	2.32	958	796	83.0%	78	57	73.1%	68	49	72.1%	122	84	68.9%
Overall	222.46	2,774,649	1,035,884	37.3%	1,998	1,088	54.5%	1,070	550	51.4%	12,701	7,946	62.6%

stream I may trigger. If no signal or exception is triggered, the value of Sig is 0.

Given the CPU emulator E, the real device R, our differential testing engine guarantees that E's initial state $CPU_I(E)$ is equal to R's initial state $CPU_I(R)$. $CPU_I(E) = CPU_I(R)$ iff:

$$\forall \phi \in < PC, Reg, Mem, Sta >: \phi_E = \phi_R$$

After the execution of I, I is treated as an inconsistent instruction stream if the final state $CPU_F(E)$ is not equal to the R's final state $CPU_F(R)$. More formally, $CPU_F(E) \neq CPU_F(R)$ iff:

$$\exists \phi \in [PC, Reg, Mem, Sta, Sig] : \phi_E \neq \phi_R$$

3.2.2 Our Strategy. To conduct the differential testing, we insert prologue and epilogue instructions. We first register the signal handlers to capture different signals. To make the initial state consistent, we set the value of general purpose registers to zero except PC. After setting up the initial state, an instruction stream will be executed. Then we dump the CPU state either after the execution or in the signal handler so that we can compare the execution result. For registers including status register (i.e., APSR), we push them on the stack and then write them into a file. For the memory, we utilize Capstone [10] to extract the target memory address that the instruction will be written into. After that, we load the target address, and push it on the stack for later inspection. Note that the number of memory write instructions is limited. We manually check the effectiveness of Capstone in analyzing these instructions and find it to work well. Finally, we compare the result collected from the emulator and a real device. If the instruction stream results in a different CPU final state, ($CPU_F(E) \neq CPU_F(R)$), it will be treated as an inconsistent instruction stream.

3.3 Implementation Details

We implement EXAMINER in Python, C and ARM assembly. In particular, we implement the test case generator in Python. We parse the ASL code, extract the lexical and syntactic information with regular expressions. We use Z3 [23] as the SMT solver to solve the constraints. The differential testing engine is implemented in C and assembly code with some glue scripts in Python. Specifically, the initial state setup and the execution result dumping is implemented with inline assembly code. In total, EXAMINER contains 5, 074 lines of Python code, 220 lines of C code, and 200 lines of assembly code.

4 EVALUATION

In this section, we evaluate EXAMINER by answering the following four research questions.

- **RQ1:** Is EXAMINER able to generate sufficient test cases?
- **RQ2:** Is EXAMINER able to detect inconsistent instructions? What are the root causes of these inconsistent instructions?
- **RQ3:** Is EXAMINER general to be applied to the other emulators?
- **RQ4:** What are the possible usage scenarios of inconsistent instructions?

4.1 Sufficiency of Test Case Generator (RQ1)

We generate the test cases according to the ARMv8-A manual, which introduces ASL. Specifically, the manual includes four different instruction sets. In AArch 64 mode, the A64 instruction set is supported. For the AArch 32 mode, it consists of three different instruction sets. They are ARM32 with 32-bit instruction length (A32), Thumb-2 with instruction length of mixed 16-bits and 32-bits (T32), and Thumb-1 with 16-bit instruction length (T16). They are also supported by previous ARM architectures (e.g., ARMv5, ARMv6, ARMv7). To locate the inconsistent instructions in different ARM architectures, we generate the test cases for all the instruction sets.

The number of generated test case is sufficient. Table 2 shows the statistics of the generated instruction streams. The column "EXAMINER" denotes the number of different attributes for our test case generator. In total, 2, 774, 649 instruction streams are generated within 4 minutes, which cover 1, 998 instruction encodings in 1, 070 instructions. Note that the total number of instruction encodings and instructions in ARM manual is 1, 998 and 1, 070, respectively, which means all the instruction encodings and instructions are covered. Note that the generated instruction streams are rather small for T16 due to the small number of instruction encoding schemes and limited instruction length. Overall, all the generated instruction streams are syntactically correct, which means they all map to one of the encoding schemas. Furthermore, more than 12 thousand constraints and their negations, which are related to encoding symbols, are solved, indicating the multiple behaviors of the instructions are explored.

To further demonstrate the effectiveness of the test case generator, we randomly generate the same number of test cases for each instruction set. We repeat the randomly generated process for 10 times and calculate the average value. Then we check whether the generated instructions are syntactically correct ones or not. If

```
boolean AArch32.ExclusiveMonitorsPass(bits(32)
       address, integer size)
// It is IMPLEMENTATION DEFINED whether the
// detection of memory aborts happens before or
// after the check on the local Exclusive Monitor.
// As a result, a failure of the local monitor can
// occur on some implementations  even if the
// memory access would give an memory abort.
      ...
      return
```

Figure 5: Two different implementations are defined in the annotation of function ExclusiveMonitorsPass, which is called by many instructions' executing code

they are, we calculate how many instruction encodings, how many instructions, and how many constraints are covered by these instruction streams. According to the Column "Random" and "Ratio" in Table 2, only 37.3% generated instruction streams are syntactically correct, which means all the others are illegal instructions and they are not effective to test the potential different behaviors between real devices and CPU emulators. Among the syntactically correct instruction streams, the randomly generated instruction streams can only cover 54.5% instruction encodings and 51.4% instructions. Nearly a half of instructions can not be covered with the randomly generated instruction streams. Specifically, many of the T32 instructions cannot be covered with randomly generated instructions, which means many of these instructions have fixed values. As for the coverage of constraints, 37.4% constraints can not be explored, resulting in a relatively limited behaviors being explored.

Answer to RQ1: Examiner can generate sufficient test cases, which are all syntactical correct instruction streams and can cover all instruction encodings and instructions. On the contrary, only 37.3% of the same number of randomly generated instruction streams are syntactical correct. Furthermore, 45.5% instruction encodings, 48.6% instructions, and 37.4% constraints cannot be explored by these randomly generated instructions.

4.2 Differential Testing Results and Root Causes (RQ2)

We feed the generated test cases into our differential testing engine to locate the inconsistent instructions. Table 3 shows the result.

Experiment Setup We conduct the differential testing between QEMU (version 5.1.0) and four real devices (OLinuXino iMX233 in ARMv5, RaspberryPi Zero in ARMv6, RaspberryPi 2B in ARMv7, and Hikey 970 in ARMv8). For ARMv5, only ARM32 is supported. Meanwhile, QEMU does not support Thumb-2 for ARM1176 of ARMv6. Thus, we only test the A32 instruction set on ARMv5 and ARMv6.

In total, it takes around 2700 seconds of CPU time for QEMU, which is run on the Intel i7-9700 CPU. For the real devices, the CPU time cost ranges from 5276 seconds to 46238 seconds (13 hours), depending on the specific devices. Thanks to the representative test cases, the differential testing for all the test cases can be finished within acceptable time.

Testing Result According to Table 3, 171,858 inconsistent instruction streams are found, owning to 6.2% of the whole test cases. Note one instruction stream may be tested in different architectures (e.g.,A32 instruction set in ARMv5, ARMv6, and ARMv7), the number in column "Overall" is the union of the other columns. Furthermore, these inconsistent instruction streams cover 531 different instruction encodings and 316 instructions, owning 26.6% and 29.5% of the tested instruction encodings and instructions, respectively.

Inconsistent Behaviors We further analyze the inconsistent instruction streams and categorize them according to our modeled CPU. We noticed that most of the inconsistent streams (i.e., 95.2%) will trigger different signals between the real device and emulators. A small number of instruction streams may not trigger the signal or trigger the same signal but have different register or memory values (i.e., 4.8%). 2 instructions can make QEMU crash but are executed normally in the real devices. Thus, we categorize them as "Others".

Root Cause Based on the inconsistent streams, we explore the root cause. First, there are implementation bugs. We discovered 4 bugs in QEMU [8, 13, 16, 22] in total, which influence 11 instruction encodings. Some of the bugs are related to very common instructions. The first bug influences the `BLX` instruction [8]. The `BLX` instruction can be an undefined one in specific cases, which should raise SIGILL signal. However, QEMU does not follow the specification and will disassemble it as a FPE11 instruction. In this case, the whole execution logic is wrong. The second bug influence STR instruction [13] and is illustrated in detail in Figure 2. QEMU does not properly check the condition that the STR instruction in thumb mode can be an undefined instruction, which result in inconsistent execution results. The third bug influences many load-/store instructions [16] (e.g., LDRD, STRD, etc). The target address of these load/store instructions should be word aligned. However, QEMU does not check it properly. The last bug is about WFI instruction [22] and it can make QEMU crash. `WFI` denotes waiting for interrupt and is usually used in system-mode emulation. However, ARM manual specifies that it can also be used in user-space. QEMU does not handle this instruction well and an abort will be generated. All of the 4 bugs are confirmed and patched by QEMU developers. This also demonstrates the capability of Examiner in discovering the bugs of the emulator implementation.

Apart from the bugs, most of the inconsistent instructions are due to the undefined implementation in the ARM manual. There are three different kinds of undefined implementations. The first one is UNPREDICTABLE (Section 2.2). UNPREDICTABLE leaves open implementation decision for emulators and processors. The second is Constraint UNPREDICTABLE. Constraint UNPREDICTABLE provides candidate implementation strategies and the developer or vendor can choose from one of them. The third is that the annotation of the ASL code indicates the implementation is undefined. Figure 5 shows an example. In the function `ExclusiveMonitorsPass`, which is called by the executing code of instruction `STREXH`, there is an annotation for the implementation. Note the check on the *local Exclusive Monitor* would update the value of a register. Thus, if the detection of memory aborts happens before the check, the value of the register would not be updated while the detection happens after the check can update the value, resulting in different register value.

Table 3: The results of differential testing for QEMU. "CPU Time" denotes the sum of the CPU time for all test cases, which is in seconds. We do not count the sum of CPU time for real devices as they have different CPUs. "Inst" denotes Instruction. "Inst_S" denotes Instruction Stream. "Inst_E" denotes Instruction Encoding. UNPRE. denotes UNPREDICTABLE. X | Y : X denotes the number of the attribute indicated by the row name while Y denotes the percentage of dividing X by Z. For data in "Testing Result", Z stands for the row "Tested Inst_S", "Tested Inst_E", or "Tested Inst". For data in and "Root Cause", Z stands for "Inconsistent Inst_S", "Inconsistent Inst_E", or "Inconsistent Inst".

Architecture	ARMv5	ARMv6	ARMv7		ARMv8	Overall
Experiment Setup						
Instruction Set	A32	A32	A32	T32&T16	A64	-
QEMU Binary	qemu-arm	qemu-arm	qemu-arm		qemu-aarch64	-
QEMU Model	ARM926	ARM1176	Cortex-A7		Cortex-A72	-
Device Name	OLinuXino IMX233	RaspberryPi Zero	RaspberryPi 2B		Hikey 970	-
CPU Time (Device)	46238.0s	6901.7s	6194.2s	5276.0s	9145.0s	-
CPU Time (QEMU)	530.5s	540.6s	538.0s	462.1s	625.9s	2702.1s
Tested Inst_S	870,221	870,221	870,221	809,728	1,094,700	2,774,649
Tested Inst_E	550	550	550	609	839	1,998
Tested Inst	481	481	481	462	581	1,070
Testing Result	The percentage is based on the number of tested instructions (streams/encodings)					
Inconsistent Inst_S	40,892 | 4.7%	18,043 | 2.1%	66,860 | 7.7%	51,823 | 6.4%	21,373 | 2.0%	171,858 | 6.2%
Inconsistent Inst_E	184 | 33.5%	175 | 31.8%	273 | 49.6%	271 | 44.5%	17 | 2.0%	531 | 26.6%
Inconsistent Inst	173 | 36.0%	167 | 34.7%	232 | 48.2%	228 | 49.4%	15 | 2.6%	316 | 29.5%
Inconsistent Behaviors	The percentage is based on the number of inconsistent instructions (streams/encodings)					
Signal (Inst_S)	38,480 | 94.1%	17,635 | 97.7%	66,660 | 99.7%	50,940 | 98.3%	16,656 | 77.9%	163,659 | 95.2%
Signal (Inst_E)	175	170	268	267	15	521
Signal (Inst)	164	162	227	224	13	312
Register/Memory (Inst_S)	2,411 | 5.9%	407 | 2.3%	199 | 0.3%	881 | 1.7%	4,716 | 22.1%	8,195 | 4.8%
Register/Memory (Inst_E)	28	15	22	19	3	64
Register/Memory (Inst)	28	15	22	16	3	54
Others (Inst_S)	1 | 0.0%	1 | 0.0%	1 | 0.0%	2 | 0.0%	1 | 0.0%	4 | 0.0%
Others (Inst_E)	1	1	1	2	1	4
Others (Inst)	1	1	1	1	1	2
Root Cause	The percentage is based on the number of inconsistent instructions (streams/encodings)					
Bugs (Inst_S)	1 | 0.0%	1 | 0.0%	1 | 0.0%	582 | 1.1%	1 | 0.0%	584 | 0.3%
Bugs (Inst_E)	1	1	1	9	1	11
Bugs (Inst)	1	1	1	6	1	7
UNPRE. (Inst_S)	40,891 | 100.0%	18,042 | 100.0%	66,859 | 100.0%	51,241 | 98.9%	21,372 | 100.0%	171,274 | 99.7%
UNPRE. (Inst_E)	183	174	272	269	16	527
UNPRE. (Inst)	172	166	231	227	14	314

Note that we can feed the instruction streams into our symbolic execution engine and it will check whether an instruction stream is UNPREDICTABLE or not automatically. In this case, users can filter out the test cases whose implementations are not defined and use the filtered ones to explore the bugs of emulators. EXAMINER is proposed to find the inconsistent instructions. Thus, we include the instruction streams that can result in UNPREDICTABLE behavior as the test cases.

Answer to RQ2: EXAMINER can detect inconsistent instructions. In total, 171,858 inconsistent instruction streams are found, which covers 26.6% (i.e.,531/1998) instruction encodings and 29.5% instructions (i.e., 316/1070). The implementation bugs of QEMU and the undefined implementation in ARM manual are the major root causes. 4 bugs are discovered and confirmed by QEMU developers, which influence 11 instruction encodings including commonly used instructions (e.g., `BLX`).

4.3 Generalization of EXAMINER

To demonstrate the generality of EXAMINER, we further apply EXAMINER on evaluating the other two lightweight but also popular CPU emulators (i.e., Unicorn in version 1.0.2rc4 and Angr in version 9.0.7833). Different from QEMU, Unicorn and Angr do not provide options to specify the ARMv5 or ARMv6 architecture. In this case, we evaluate ARMv7 and ARMv8. Meanwhile, Unicorn and Angr do not have good support on advanced instructions [5]. For instance, many SIMD instructions will make Angr crash, resulting in 5 new bugs. Instructions (e.g., `WFE` [6]) that rely on kernel or multiprocessor are also not supported. Thus, we filter out these instructions in the experiment. Note that both Unicorn and Angr do not support signals. In this case, we build the mapping relationship between the exceptions raised by Angr or Unicorn and the signals triggered by operating systems. For example, the exception *SimIRSBNoDecodeError* raised by Angr maps to signal number 4, which represents an illegal instruction, triggered by operating systems.

Table 4 shows the result. 223, 264 and 120, 169 inconsistent instructions streams are identified for Unicorn and Angr, respectively. They also cover hundreds of instruction encodings. They share many of the same instruction streams with QEMU. For example, 28.2% and 21.6% instruction streams among the inconsistent instruction streams of Unicorn and Angr can also trigger inconsistent behaviors between QEMU and real devices. Similar to QEMU, the inconsistent behaviors mainly consists of two types. One is the different triggered signals and the other is the register or memory

Table 4: The results of differential testing for Unicorn and Angr. The attributes denotes the same meaning explained in the caption of Table 3.

Tool	Unicorn				Angr			
Architecture	ARMv7		ARMv8	Overall	ARMv7		ARMv8	Overall
Instruction Set	A32	T32 & T16	A64	-	A32	T32 & T16	A64	-
CPU Time	31.8s	32.9s	32.4s	97.1s	7654.2s	7873.1s	10004.1s	25531.4s
Tested Inst_S	328,780	336,987	371,770	1,037,537	328,780	336,987	371,770	1,037,537
Tested Inst_E	352	398	205	955	352	398	205	955
Tested Inst	313	285	77	418	313	285	77	418
Testing Result	The percentage is based on the number of tested instructions (streams/encodings)							
Inconsistent Inst_S	103,520 | 31.5%	119,394 | 35.4%	350 | 0.1%	223,264 | 21.5%	70,493 | 21.4%	37,364 | 11.1%	12,312 | 3.3%	120,169 | 11.6%
Inconsistent Inst_E	267 | 75.9%	300 | 75.4%	3 | 1.5%	570 | 59.7%	154 | 43.8%	161 | 40.4%	23 | 11.2%	338 | 35.4%
Inconsistent Inst	231 | 73.8%	254 | 89.1%	2 | 2.6%	298 | 71.3%	126 | 40.3%	130 | 45.6%	10 | 13.0%	197 | 47.1%
Intersection with QEMU	The percentage is based on the number of inconsistent instructions (streams/encodings)							
Inconsistent Inst_S	39,515 | 38.2%	23,146 | 19.4%	350 | 100.0%	63,011 | 28.2%	22,240 | 31.5%	3,740 | 10.0%	0 | 0.0%	25,980 | 21.6%
Inconsistent Inst_E	169 | 63.3%	166 | 55.3%	3 | 100.0%	338 | 59.3%	114 | 74.0%	75 | 46.6%	0 | 0%	189 | 55.9%
Inconsistent Inst	161 | 69.7%	161 | 63.4%	2 | 100.0%	199 | 66.8%	88 | 69.8%	47 | 36.1%	0 | 0%	101 | 51.3%
Inconsistent Behaviors	The percentage is based on the number of inconsistent instructions (streams/encodings)							
Signal (Inst_S)	103,514 | 100.0%	118,141 | 99.0%	350 | 100.0%	222,005 | 99.4%	70,487 | 100.0%	37,357 | 100.0%	12,312 | 100.0%	120,156 | 100.0%
Signal (Inst_E)	266	299	3	568	154	161	23	338
Signal (Inst)	230	253	2	297	126	130	10	197
Register/Memory (Inst_S)	6 | 0.0%	1,253 | 1.0%	0 | 0.0%	1,259 | 0.6%	6 | 100%	7 | 0.0%	0 | 0.0%	13 | 0.0%
Register/Memory (Inst_E)	1	5	0	6	1	2	0	3
Register/Memory (Inst)	1	5	0	5	1	2	0	2
Root Cause	The percentage is based on the number of inconsistent instructions (streams/encodings)							
Bugs (Inst_S)	0 | 0.0%	529 | 0.4%	0 | 0.0%	529 | 0.2%	0 | 0.0%	0 | 0.0%	0 | 0.0%	0 | 0.0%
Bugs (Inst_E)	0	7	0	7	0	0	0	0
Bugs (Inst)	0	5	0	5	0	0	0	0
UNPRE. (Inst_S)	103,520 | 100%	118,865 | 99.6%	350 | 100%	222,735 | 99.8%	70,493 | 100%	37,364 | 100%	12,312 | 100%	120,169 | 100%
UNPRE. (Inst_E)	267	296	3	566	154	161	23	338
UNPRE. (Inst)	231	253	2	297	126	130	10	197

Table 5: The statistics on detecting emulators.

Mobile Type	CPU	A64	A32	T32 & T16
Samsung S8	SnapDragon 835	✓	✓	✓
Huawei Mate20	Kirin 980	✓	✓	✓
IQOO Neo5	SnapDragon 870	✓	✓	✓
Huawei P40	Kirin 990	✓	✓	✓
Huawei Mate40 Pro	Kirin 9000	✓	✓	✓
Honor 9	Kirin 960	✓	✓	✓
Honor 20	Kirin 710	✓	✓	✓
Blackberry Key2	SnapDragon 660	✓	✓	✓
Google Pixel	SnapDragon 821	✓	✓	✓
Samsung Zflip	SnapDragon 855	✓	✓	✓
Google Pixel3	SnapDragon 845	✓	✓	✓

```
1  void sig_handler(int signum) {
2      record_execution_result(i++);
3      siglongjmp(sig_env, i);
4  }
5
6  Bool JNI_Function_Is_In_Emulator() {
7      register_signals(sig_handler);
8      i = sigsetjmp(sig_env,0);
9      switch (i){
10         case 1:
11             execute(inconsistent_instruction_n);
12             record_execution_result(i++);
13             longjmp(sig_env,i++);
14         case 2:
15             ...
16         case n:
17     }
18     return compare_result();
19 }
```

Figure 6: Pseudo code of the native code for detecting the emulator.

values. We also explored the root cause of these inconsistent instructions. Similar to QEMU, undefined implementation and bugs are the major causes. 3 bugs are located in Unicorn.

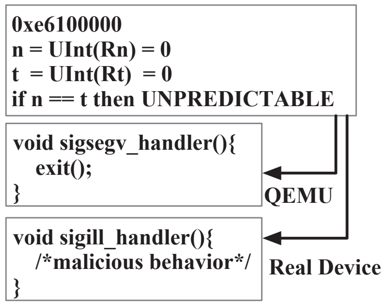
Answer to RQ3: Examiner is general to be applied to the other CPU emulators (i.e., Unicorn and Angr). With Examiner, we disclosed 8 more bugs (5 in Angr and 3 in Unicorn) and located a huge number of inconsistent instruction streams in the two CPU emulators).

4.4 Applications of Inconsistent Instructions (RQ4)

The inconsistent instructions can be used to detect the existence of emulators. Furthermore, detecting emulators can prevent the binary from being analyzed or fuzzed, which is known as anti-emulation and anti-fuzzing technique.

4.4.1 Emulator Detection. The inconsistent instructions can be used to detect emulators. Considering the popularity of Android systems, we target Android applications. Specifically, we build a native library by using the inconsistent instructions.

Figure 6 shows the pseudo code of the library. Function *JNI_Function_Is_In_Emulator* (line 6) returns True if the emulator is detected. Inside the function, we register signal handlers for different signals (line 7). After the execution of each instruction stream, we will record the execution result either in the signal handler (line 2) or after the execution (line 12). Then we use the function *longjmp* (line 13) or *siglongjmp* (line 3) to jump back to the place where calling *sigsetjmp* (line 8). As *i* would increase by 1 after the execution of one instruction stream, we can execute hundreds of instruction



```
0xe6100000
n = UInt(Rn) = 0
t  = UInt(Rt)  = 0
if n == t then UNPREDICTABLE

void sigsegv_handler(){
    exit();
}

void sigill_handler(){
    /*malicious behavior*/
}
```

Figure 7: Inconsistent instruction can prevent the malicious behavior being detected by emulators

```
0x10000: e51b3008 LDR r3,[fp,#-8]
0x10004: e1a03000 MOV r3,r0
0x10008: e7cf0e9f BFC r0, #0xf, #1
// BFC instruction is to clear specific bits
// e7cf0e9f is an UNPREDICTABLE encoding
// e7cf0e9f is executed normally in real device
// e7cf0e9f triggers SIGILL signal on QEMU
0x1000c: e1a00003 MOV r0,r3
0x10010: e50b3008 STR r3,[fp,#-8]
```

Figure 8: Instrumented instruction streams for anti-fuzzing.

streams in one function by adding corresponding *case* conditions. Each instruction stream can make an equal contribution to the final decision on whether the current execution environment is in real devices or emulators. Finally, if more instruction streams decide the application are running inside an emulator, the *compare_result()* will return True and vice versa.

We automatically generate the test library with template code and build three Android apps for different instruction set (i.e., A64, A32, and T32 & T16). We run the applications on 12 different mobiles in different CPUs from 6 different vendors. Meanwhile, we run the applications in the Android emulator provided by Android studio (version 4.1.2). If the function *JNI_Function_Is_In_Emulator* returns True in the emulator and returns False in real mobiles. We consider it to successfully detect the emulator. Table 5 shows the evaluation result, all the mobile apps can detect the existence of emulators and real mobiles successfully.

4.4.2 *Anti-Emulation.*

Anti-emulation technique is important. On the attacker's side, it can be proposed to increase the bar for analyzing the malware. On the defender's side, commercial software needs to protect the core functionality and algorithms from being analyzed. Thus, it is widely used in the wild [65].

The inconsistent instructions can be used to conduct anti-emulation and can prevent the malware's malicious behavior from being analyzed. Specifically, we use one of the state-of-the-art dynamic analysis platforms (i.e., PANDA [11]) to demonstrate the usage. PANDA is built upon QEMU and supports many functionalities (e.g., taint analysis, record and replay). We port one of the open source rootkits (i.e., Suterusu [14]) to Debian 7.3. We register two different signal handlers for SIGILL and SIGSEGV, respectively. Then we instrument one instruction stream (i.e., 0xe61000000). This is an LDR instruction. According to the encoding schema, encoding symbol `Rn` and `Rt`' values are both zero. The ASL code of decoding would check whether n equals to t. If so, it is an UNPREDICTABLE instruction stream. Real devices think this is an illegal instruction stream and will raise the SIGILL signal while PANDA tries to execute the instruction stream. Then SIGSEGV will be raised as the address pointed by R0 cannot be accessed. In this case, the malicious behavior will only be triggered in real devices. Meanwhile, when we use the PANDA to analyze the malware, no malicious behavior will be monitored and the program will exit inside the *sigsegv_handler*.

Table 6: Overhead information of anti-fuzzing.

Library	Test Suite[1]	Space Overhead	Runtime Overhead
libpng (readpng)	built-in (254)	4.0% (+7KB)	0.52%
libjpeg (djpeg)	GIT (97)	4.3% (+8KB)	0.61%
libtiff (tiffinfo)	built-in (61)	2.2% (+8KB)	0.59%
Overall		3.5%	0.57%

[1] The number of test inputs in test suite is shown in the bracket.

4.4.3 *Anti-Fuzz.*

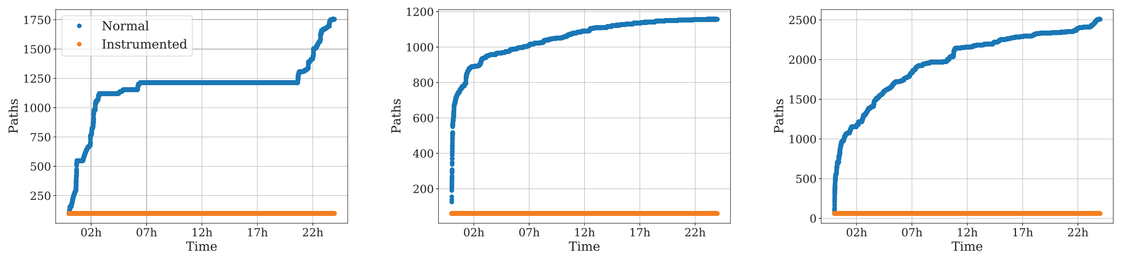
Fuzzing is widely used to explore vulnerabilities. To help the released binaries from being fuzzed by attackers, researchers utilize anti-fuzzing techniques [35, 43]. Considering that many new binary fuzzing frameworks are based on QEMU [2, 30, 33, 67], the inconsistent instructions can be used as a mitigation approach towards fuzzing technique.

We demonstrate how the inconsistent instructions can be used to conduct anti-fuzzing tasks with a relatively low overhead and high decreased coverage ratio. Specifically, we instrument a snippet of assembly code into the release binary, which is shown in Figure 8. At address 0x10008, the instruction `BFC` is used to clear bits for register R0. Note we move the value of R0 to R3 before the instruction `BFC` and return it back after the execution of `BFC`. This can guarantee the instrumented instructions will not affect the execution result of the binary on the real device. The instruction stream 0xe7cf0e9f is an UNPREDICTABLE one. It can be executed normally in real devices while triggering a signal on QEMU.

We developed a GCC plugin to instrument the above mentioned inconsistent instruction streams at each function entry and apply this plugin on three popular used libraries (i.e., libtiff, libpng, and libjpeg) during the compilation process to generate released binaries. Table 6 shows the space and runtime overhead of the instrumented binary compared with the normal (non-instrumented) ones. The space overhead is measured by comparing the binary size. For runtime overhead, we measure it by running test suites on both binaries and comparing the cost of time. We noticed that the instrumented binary imposes negligible space and runtime overhead to the binary. The average space overhead for the protected binary is around 4%, and the runtime overhead is less than 1%.

We then measure the effectiveness of anti-fuzzing. We fuzz the instrumented binaries and the normal ones with AFL-QEMU (version 2.56b) for 24 hours. The seed corpus is the test suite used for each library in Table 6. We collect the coverage information for the instrumented and the normal ones. Figure 9 shows the results. It is easy to see that the coverage for instrumented binaries cannot increase (because QEMU fails to execute binaries correctly), while the normal ones will increase with the fuzzing time.

Note this is to demonstrate the ability of inconsistent instructions on anti-fuzzing tasks. How to stealthily use these instructions is



(a) libjpeg (b) libpng (c) libtiff

Figure 9: The result of Anti-Fuzzing experiment on three libraries. The blue lines show the coverage over 24 hours of fuzzing. The orange line shows the coverage for instrumented binaries, which cannot increase due to failed executions of QEMU.

out of our scope. It is not easy for attackers to precisely recognize all the inconsistent instructions, which will be discussed in detail (Section 5).

Answer to RQ4: The inconsistent instructions are useful. We demonstrate that the inconsistent instructions can be used to detect the existence of the CPU emulator and prevent the malicious behavior from being monitored by dynamic analysis frameworks. Furthermore, the path coverage of programs fuzzed in emulators can be highly decreased with the help of inconsistent instructions.

5 DISCUSSION

Advancement over iDEV [57] Though both Examiner and iDEV use differential testing with generated test cases, Examiner has better scalability and capability on locating inconsistent instruction streams in terms of the following perspectives. 1) **Test case generation**: Examiner utilizes the symbolic execution technique to generate the test cases, which can cover more execution paths. The fact that we can detect the emulator bugs with about 2.7 million instruction streams demonstrates the effectiveness of our test cases. On the contrary, 34 million instruction streams are tested by iDEV, and no bugs are found. 2) **Differential testing**: iDEV only compares the triggered signals while Examiner compares the whole CPU state including signal number or raised exceptions, register value, memory value, etc. In this case, we can find more inconsistent instructions compared with iDEV in theory. For instance, among the 171,858 inconsistent instruction streams for QEMU, 8,195 are inconsistent in terms of different register or memory values, which cannot be detected by iDEV. Furthermore, Unicorn and Angr can not trigger the signals and iDEV can not work on testing these two emulators. Thus, the identified 223,264 instruction streams for Unicorn and the 120,169 ones for Angr can not be detected by iDEV in theory, either. Examiner supports testing Unicorn and Angr by building the mapping relationship between the triggered signal number by real devices and the raised exceptions by the emulators. 3) **Evaluation**: We evaluate Examiner on 4 different ARM versions and three CPU emulators while iDEV only evaluate QEMU on one specific ARM version (i.e., ARMv7). This demonstrates the scalability of Examiner. For iDEV, testing 34 millions test cases on machine in ARMv5&v6 would take a rather long time (i.e., more than 500 CPU hours), which is not efficient. Thanks to our symbolic execution engine, we can explore most of the execution paths with about 2.7 million test cases, which can save a lot of testing time, and find bugs on all the emulators. 4) **Usage Scenario**: Although the iDEV authors discussed the potential usage scenario of the inconsistent instructions, we demonstrate how these inconsistent instruction can be used in practice and how they can be abused by attackers with three different applications.

Detecting (Ab)Used Inconsistent Instructions Section 4.4 shows that attackers or vendors can (ab)use these inconsistent instructions. The abused inconsistent instructions are not easy to be detected. This is because there are many inconsistent instructions and some of them are even commonly used (i.e., `BLX`). Apart from this, attackers can encrypt these instruction streams as data. Then these encrypted instruction streams can be decrypted and executed during runtime, which can increase the bar for detection. Furthermore, how to hide these inconsistent instruction streams from being detected is a *Cat and Mouse* problem. Stealthily using these instructions is out of our scope.

Testing Instructions in Privileged Environments Currently, the generated instruction streams are tested under unprivileged mode in both CPU emulators and real devices. Some instruction streams may have different execution results under privileged mode. For instance, the instruction `WFI`, which results in a bug of QEMU user-mode, may not be an inconsistent instruction while executing in privileged mode. We plan to port Examiner to kernel-space in the future.

Testing Instruction Stream Sequences Examiner now tests only one instruction stream each time during the differential testing. We can also test multiple instruction streams (instruction stream sequences) in the differential testing. The instruction stream sequences may trigger multiple system states and we can test the decoding/executing logic towards different state flags. How to design representative instruction stream sequences, and how to locate the inconsistent one will be the challenge, which is left as future work. Nevertheless, we have already discovered a huge number of inconsistent instruction streams with Examiner, covering 29.5% of instructions. Every instruction stream sequence that contain the inconsistent instruction stream can result in inconsistent behaviors.

Other Architectures The whole framework of EXAMINER is architecture-independent. We apply symbolic execution technique on ARM ASL, which can help to explore multiple behaviors and generate sufficient test cases automatically. For the other architectures, symbolic execution technique can also be used if similar architecture specific language is provided. Otherwise, new test case generation algorithm should be developed in order to explore more execution behaviors. However, this is one time effort. The generated test cases can be used to test the implementation of both hardwares and emulators. In addition, the CPU state for the other architectures should also be modeled correctly. Based on the correctly modeled CPU state, the differential testing engine needs to set the initial CPU state before the execution of the target instruction and dump the CPU state for comparison after the execution.

6 RELATED WORK

6.1 Testing CPU Emulators

Several works are proposed to test the CPU emulators. Lorenzo et al. proposed EmuFuzzer to test the CPU emulators [54, 55]. However, the seed used for testing mainly relies on randomization and a CPU-assisted mechanism, which may not cover all the CPU behaviors. KEmuFuzzer is proposed to test the whole system emulators [53]. However, KEmuFuzzer relies on the manually written template to generate test cases. PokeEMU [52] utilizes binary symbolic execution to generate more test cases from a high-fidelity emulator and apply these test cases on low-fidelity emulators. However, whether the high-fidelity emulator strictly follows the rule of specification is unknown. Furthermore, all the above mentioned works target x86/x64 architectures. Though iDEV [57] studies the semantic deviation problem in ARM instruction, the generated test cases are not sufficient and redundant, which cannot cover all the instruction behaviors. Meanwhile, iDEV only focuses on the triggered signals during the execution process without checking the whole CPU state, resulting in many inconsistent instructions unexplored. Furthermore, the evaluation is limited to ARMv7 and QEMU. There are many other ARM architectures (e.g., ARMv5, ARMv6, and ARMv8) and lightweight but also popular emulators (i.e., Unicorn, Angr), which many frameworks are based on [25, 37, 51, 64].

6.2 Differential Testing

Differential testing is introduced by McKeeman et al. [56] to detect bugs by comparing the inconsistent behaviors between different entities. For example, Yang et al. proposed Csmith, a powerful tool that can generate multiple C programs. With Csmith, hundreds of bugs are detected in the C compiler. Regarding the same goal, Le et al. introduced equivalence modulo inputs (EMI) [47] and many other differential testing tools are built based on EMI to validate the compiler implementations [48, 60]. In addition, researchers also utilize differential testing to validate the Database Management Systems (DBMS). Slutz et al. proposed the tool RAGS to explore bugs by executing different SQL queries on multiple DBMS. Gu et al. evaluate the accuracy of DBMS optimizer by using options and hints to force the generation of different query plans. Jung et al. developed APOLLO [42] to test the performance regression bugs in DBMSs . Furthermore, differential testing is powerful and applied to different domains such as testing SMT solvers [62, 63], JVM implementations [44] , symbolic execution engines [44], and PDF readers [46].

6.3 Anti-Emulation Technique

Previous anti-emulation works [58] divide the anti-emulation technique into three categories. They are differences in behavior, differences in timing, and hardware specific values. Our work can automatically locate the inconsistent instructions, which result in different behaviors and can be used by the previous anti-emulation technique. Jang et al. [39] address the importance of anti-emulation techniques on protecting the Commercial-Off-the-Shelf (COTS) software from being debugged or used without buying hardware. They propose three different anti-emulation techniques. However, some techniques rely on the race condition are not easy to trigger.

7 CONCLUSION

We design and implement EXAMINER, a framework that can automatically locate the inconsistent ARM instructions. With EXAMINER, we generate 2,774,649 representative instruction streams and detect 171, 858 inconsistent ones for QEMU. To demonstrate EXAMINER's generalization, we further apply EXAMINER on two other emulators (i,e., Unicorn and Angr) and a huge number of inconsistent instructions are located. We noticed that bugs and undefined implementation in ARM manual are the root causes. Furthermore, we disclosed 12 bugs (4 in QEMU, 3 in Unicorn, 5 in Angr). Some of them influence commonly used instructions (e.g., `BLX`) and can even crash the emulators (e.g., QEMU and Angr). We also demonstrate the capability of inconsistent instructions on detecting emulators, anti-emulation, and anti-fuzzing.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their comments that greatly helped improve the presentation of this paper. We also want to thank Dr. Manuel Rigger for shepherding our paper. This work was partially supported by the National Natural Science Foundation of China (NSFC) under Grant 61872438, Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (2018R01005), the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform), HK RGC Project (No. PolyU 152239/18E).

REFERENCES

[1] 64 Bit Juno r2 ARM® Development Platform. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Juno%20r2%20datasheet.pdf.
[2] AFL QEMU Mode: high-performance binary-only instrumentation for afl-fuzz. https://github.com/google/afl/tree/master/qemu_mode.
[3] Angr. https://angr.io/.
[4] ARM Exploration tools. https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools.
[5] ARM SIMD Instructions. https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/What-is-SIMD-/ARM-SIMD-instructions.
[6] ARM WFE Instruction. https://developer.arm.com/documentation/ddi0360/e/programmer-s-model/additional-instructions/wait-for-event-wfe.
[7] AttributeError Bug in Angr. https://github.com/angr/angr/issues/2803.
[8] BLX instruction bug in QEMU. https://bugs.launchpad.net/qemu/+bug/1925512.
[9] Bugs in Unicorn. https://github.com/unicorn-engine/unicorn/issues/1424.
[10] Capstone. https://www.capstone-engine.org/.
[11] PANDA.re. https://panda.re/.
[12] QEMU. https://www.qemu.org/.
[13] STR instruction bug in QEMU. https://bugs.launchpad.net/qemu/+bug/1922887.
[14] Suterusu. https://github.com/mncoppola/suterusu.

[15] TriforceAFL. https://github.com/nccgroup/TriforceAFL.
[16] Unaligned data access bug in QEMU. https://bugs.launchpad.net/qemu/+bug/1905356.
[17] Unicorn. https://www.unicorn-engine.org/.
[18] VABS Bug in Angr. https://github.com/angr/angr/issues/2808.
[19] VCVT Bug in Angr. https://github.com/angr/angr/issues/2829.
[20] VMAX Bug in Angr. https://github.com/angr/angr/issues/2809.
[21] VMUL Bug in Angr. https://github.com/angr/angr/issues/2810.
[22] WFI instruction bug in QEMU. https://bugs.launchpad.net/qemu/+bug/1926759.
[23] Z3Prover. https://github.com/Z3Prover/z3.
[24] Abdulla Alwabel, Hao Shi, Genevieve Bartlett, and Jelena Mirkovic. 2014. Safe and automated live malware experimentation on public testbeds. In *Proceedings of the 7th Workshop on Cyber Security Experimentation and Test.*
[25] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. 2017. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Proceedings of the 38th IEEE Symposium on Security and Privacy.*
[26] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. 2016. Extract Me If You Can: Abusing PDF Parsers in Malware Detectors.. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium.*
[27] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium.*
[28] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices* (2011).
[29] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. 2019. One Engine To Serve'em All: Inferring Taint Rules Without Architectural Semantics.. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium.*
[30] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *Proceedings of the 29th USENIX Security Symposium.*
[31] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding linux malware. In *Proceedings of the 2018 IEEE symposium on security and privacy.*
[32] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. 2019. DECAF++: Elastic whole-system dynamic taint analysis. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses.*
[33] Bo Feng, Alejandro Mera, and Long Lu. 2019. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *Proceedings of the 29th USENIX Security Symposium.*
[34] Qian Feng, Aravind Prakash, Heng Yin, and Zhiqiang Lin. 2014. Mace: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th annual computer security applications conference.*
[35] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. 2019. Antifuzz: Impeding fuzzing audits of binary executables. In *Proceedings of the 28th USENIX Security Symposium.*
[36] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis.*
[37] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin RB Butler. 2017. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.*
[38] Anoirel Issa. 2012. Anti-virtual machines and emulations. *Journal in Computer Virology* (2012).
[39] Daehee Jang, Yunjong Jeong, Sungman Lee, Minjoon Park, Kuenhwan Kwak, Donguk Kim, and Brent Byunghoon Kang. 2019. Rethinking anti-emulation techniques for large-scale software deployment. *Computers & Security* (2019).
[40] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. 2010. Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction. *ACM Transactions on Information and System Security* (2010).
[41] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. 2021. Jetset: Targeted Firmware Rehosting for Embedded Systems. In *Proceedings of the 30th USENIX Security Symposium.*
[42] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment* (2019).
[43] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. 2019. Fuzzification: Anti-fuzzing techniques. In *Proceedings of the 28th USENIX Security Symposium.*
[44] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering.*
[45] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. 2020. FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis. In *Proceedings of the 2020 Annual Computer Security Applications Conference.*
[46] Tomasz Kuchta, Thibaud Lutellier, Edmund Wong, Lin Tan, and Cristian Cadar. 2018. On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in PDF readers and files. *Empirical Software Engineering* (2018).
[47] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices* (2014).
[48] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. *ACM SIGPLAN Notices* (2015).
[49] Cătălin Valeriu Liţă, Doina Cosovan, and Dragoş Gavriluţ. 2018. Anti-emulation trends in modern packers: a survey on the evolution of anti-emulation techniques in UPA packers. *Journal of Computer Virology and Hacking Techniques* (2018).
[50] Lannan Luo, Yu Fu, Dinghao Wu, Sencun Zhu, and Peng Liu. 2016. Repackage-proofing android apps. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.*
[51] Dominik Maier, Benedikt Radtke, and Bastian Harren. 2019. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *Proceedings of the 13th USENIX Workshop on Offensive Technologies.*
[52] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems.*
[53] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. Testing system virtual machines. In *Proceedings of the 19th international symposium on software testing and analysis.*
[54] Lorenzo Martignoni, Roberto Paleari, Alessandro Reina, Giampaolo Fresi Roglia, and Danilo Bruschi. 2013. A methodology for testing CPU emulators. *ACM Transactions on Software Engineering and Methodology* (2013).
[55] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU emulators. In *Proceedings of the eighteenth international symposium on Software testing and analysis.*
[56] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* (1998).
[57] Shisong Qin, Chao Zhang, Kaixiang Chen, and Zheming Li. 2021. iDEV: exploring and exploiting semantic deviations in ARM instruction processing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis.*
[58] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. 2007. Detecting system emulators. In *Proceedings of the 2007 International Conference on Information Security*. Springer.
[59] Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Proceedings of the 16th Formal Methods in Computer-Aided Design.*
[60] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.*
[61] Jinpeng Wei, Lok K Yan, and Muhammad Azizul Hakim. 2015. Mose: Live migration based on-the-fly software emulation. In *Proceedings of the 31st Annual Computer Security Applications Conference.*
[62] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. *Proceedings of the ACM on Programming Languages* OOPSLA (2020).
[63] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.*
[64] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R Lyu. 2017. Concolic execution on small-size binaries: Challenges and empirical study. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.*
[65] Xu Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the 38th IEEE International Conference on Dependable Systems and Networks.*
[66] Lok Kwong Yan and Heng Yin. 2012. Droidscope: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security Symposium.*
[67] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *Proceedings of the 28th USENIX Security Symposium.*