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 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 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). With the inconsistent instructions, we build three security applications and demonstrate the capability of these instructions on detecting emulators, anti-emulation, and anti-fuzzing.

CCS CONCEPTS

• Security and privacy → Software security engineering.

KEYWORDS

Emulator, Differential Testing, Inconsistent Instructions

ACM Reference Format:


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]. 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
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., Emu-Fuzzer [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 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 233, 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 differentiating 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 t) for offset [31:20] and [7:7]. The values in other offsets include 6 encoding symbols and they are Rn, Rt, P, U, and Imm8.

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

(b) The ASL code for decoding the instruction.

(c) The ASL code for executing the instruction.

Fig. 1: A motivation example.

---

Apologies for the overlap in the document. The text is not visible due to the overlap. The text might include details about testing the encoded instructions for consistency, generating test cases, and the usage scenarios of the Examiner system.
the root cause of them and how they can be (ab)used.

We analyze the identified inconsistent instructions to understand the 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: encoder, decorder, product on the instructions
Output: The generated test cases

for Symbols, Constants, Constraints = ParseASL_encoder, decoder, executor
for S in Symbols do
    $S MutationSet = InitSet(S)$
for C in Constants do
    $C MutationSet = [ConstantValue]$
for C in Constraints + Negated Constraints do
    $ValueSet = SolveConstraint(, Symbols, decoder, executor)$
    if $V not in S MutationSet$ then
        $S MutationSet add V$
$MutationSets = [S MutationSet + C MutationSet]$
$TestCases = CartesianProduct(MutationSets)$
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 Constants 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).

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

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.
As for the immediate value, the symbol name is usually the ASL code. Specifically, we assign symbolic values for encoding immediate values.

### 3.1.2 Solve Constraints

For each symbol, the values for one symbol is not realistic because immediate values represent the bit length of the symbol. Note that enumerating all possibilities set to cover such cases. For the immediate value, the maximum instruction encodings cannot be 15. We include it in the mutation set to cover such cases. For the immediate value, the maximum instruction encodings cannot be 15. We include it in the mutation set to cover such cases.

We infer the type based on the symbol name. For instance, a symbol represents an 8-bit immediate value.

### 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’s initial state $CPU_I(T)$ with the tuple $<PC_T, Reg_T, Mem_T, Sta_T, Sig_T>$. $PC$ 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-consuming. $Sta$ denotes the status register, which is $APSR$ in ARM architecture. We denote the CPU’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.

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Time (s) of Examiner</th>
<th>Instruction Stream</th>
<th>Instruction Encoding</th>
<th>Instruction</th>
<th>Covered Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EXAMINER</td>
<td>Random</td>
<td>Ratio</td>
<td>EXAMINER</td>
<td>Random</td>
</tr>
<tr>
<td>A64</td>
<td>70.51</td>
<td>1,094,700</td>
<td>421,645</td>
<td>38.5%</td>
<td>839</td>
</tr>
<tr>
<td>A32</td>
<td>75.05</td>
<td>870,221</td>
<td>578,845</td>
<td>66.5%</td>
<td>550</td>
</tr>
<tr>
<td>T32</td>
<td>74.58</td>
<td>808,770</td>
<td>34,598</td>
<td>4.2%</td>
<td>531</td>
</tr>
<tr>
<td>T16</td>
<td>2.32</td>
<td>958</td>
<td>796</td>
<td>83.0%</td>
<td>78</td>
</tr>
<tr>
<td>Overall</td>
<td>222.46</td>
<td>2,774,649</td>
<td>1,035,884</td>
<td>37.3%</td>
<td>1,998</td>
</tr>
</tbody>
</table>

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

Given the CPU emulator $E$, the real device $R$, our differential testing engine guarantees that $E$’s initial state $CPU(E)$ is equal to $R$’s initial state $CPU(R)$. $CPU(E) = CPU(R)$ 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.
We feed the generated test cases into our differential testing engine within acceptable time. For the real devices, the CPU ARMv6.

As for the coverage of constraints, 37.4% constraints can be explored. According to the Column “Random” and “Ratio” in Table 2, only 37.3% of the same number of randomly generated instruction streams can only cover 54.5% instruction encodings and 51.4% instructions. Nearly a half of instructions cannot 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 syntactically 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 syntactically 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 Y. 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".

<table>
<thead>
<tr>
<th>Architecture</th>
<th>ARMv5</th>
<th>ARMv6</th>
<th>ARMv7</th>
<th>ARMv8</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experiment Setup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction Set</td>
<td>A32</td>
<td>A32</td>
<td>T32&amp;T16</td>
<td>A64</td>
<td>-</td>
</tr>
<tr>
<td>QEMU Binary</td>
<td>qemu-arm</td>
<td>qemu-arm</td>
<td>qemu-arm</td>
<td>qemu-aarch64</td>
<td>-</td>
</tr>
<tr>
<td>QEMU Model</td>
<td>ARM926</td>
<td>ARM1176</td>
<td>Cortex-A7</td>
<td>Cortex-A72</td>
<td>-</td>
</tr>
<tr>
<td>Device Name</td>
<td>OLimaXino IMX233</td>
<td>Raspberry Pi Zero</td>
<td>Raspberry Pi 2B</td>
<td>Hikey 970</td>
<td>-</td>
</tr>
<tr>
<td>CPU Time (Device)</td>
<td>46238.0s</td>
<td>6901.7s</td>
<td>6194.2s</td>
<td>5276.0s</td>
<td>9145.0s</td>
</tr>
<tr>
<td>CPU Time (QEMU)</td>
<td>530.5s</td>
<td>540.6s</td>
<td>538.0s</td>
<td>462.1s</td>
<td>625.9s</td>
</tr>
<tr>
<td>Tested Inst_E</td>
<td>879,221</td>
<td>879,221</td>
<td>879,221</td>
<td>809,728</td>
<td>1,049,700</td>
</tr>
<tr>
<td>Tested Inst</td>
<td>750</td>
<td>550</td>
<td>550</td>
<td>690</td>
<td>1,998</td>
</tr>
<tr>
<td>Tested Inst_E</td>
<td>481</td>
<td>481</td>
<td>481</td>
<td>462</td>
<td>581</td>
</tr>
<tr>
<td>Tested Inst_S</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1,070</td>
</tr>
</tbody>
</table>

Testing Result: The percentage is based on the number of tested instruction streams (instructions/encodings).

| Inconsistent Inst_S | 46,892 | 4.7% | 18,043 | 2.1% | 66,860 | 7.7% | 96,860 | 10.0% |
| Inconsistent Inst_E | 184 | 33.5% | 175 | 31.8% | 273 | 49.6% | 271 | 44.5% |
| Inconsistent Inst | 173 | 36.0% | 167 | 34.7% | 232 | 42.8% | 228 | 45.4% |

Inconsistent Behaviors: The percentage is based on the number of inconsistent instruction streams (instructions/encodings).

| Signal (Inst_S) | 36,469 | 94.1% | 17,634 | 97.7% | 66,860 | 99.7% | 51,241 | 98.9% |
| 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% | 497 | 2.3% | 199 | 0.3% | 881 | 1.7% |
| Register/Memory (Inst_E) | 28 | 15 | 22 | 19 | 3 | 64 |
| Others (Inst_S) | 1 | 1 | 10 | 9 | 3 | 54 |
| 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 instruction streams (instructions/encodings).

| Bugs (Inst_S) | 1 | 1 | 1 | 0.0% | 582 | 11.0% | 1 | 0.0% | 584 | 0.3% |
| Bugs (Inst_E) | 1 | 1 | 1 | 9 | 1 | 11 |
| Bugs (Inst_E) | 1 | 1 | 1 | 1 | 6 | 7 |
| UNPRE. (Inst_S) | 1 | 0.0% | 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 1 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 multiprocessors 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 instruction 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.
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.

---

**Table 5:** The statistics on detecting emulators.

<table>
<thead>
<tr>
<th>Mobile Type</th>
<th>CPU</th>
<th>A64</th>
<th>A32</th>
<th>T32 &amp; T16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huawei Mate20</td>
<td>Kirin 980</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>IQOO Neo5</td>
<td>Snapdragon 870</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Huawei P40</td>
<td>Kirin 990</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Huawei Mate40 Pro</td>
<td>Kirin 900</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Honor 9</td>
<td>Kirin 960</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Honor 20</td>
<td>Kirin 710</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Blackberry Key2</td>
<td>Snapdragon 660</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Google Pixel</td>
<td>Snapdragon 821</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Samsung ZFlip</td>
<td>Snapdragon 855</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Google Pixel2</td>
<td>Snapdragon 845</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

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.

#### 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
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`.

### 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 \([35, 45]\). 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 that many new binary fuzzing frameworks are based on QEMU. The instruction stream 0xe70e9f 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
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).
Unaligned data access bug in QEMU. https://bugs.launchpad.net/qemu/+bug/1926759.


WFI instruction bug in QEMU. https://bugs.launchpad.net/qemu/+bug/1926759.


WFI instruction bug in QEMU. https://bugs.launchpad.net/qemu/+bug/1926759.


WFI instruction bug in QEMU. https://bugs.launchpad.net/qemu/+bug/1926759.