EXAMINER-PRO: Testing Arm Emulators across Different Privileges

Muhui Jiang, Xiaoye Zheng, Rui Chang, Yajin Zhou, Xiapu Luo

***Abstract*—Emulators are commonly employed to construct dynamic analysis frameworks due to their ability to perform fine-grained tracing, monitor full system functionality, and run on diverse operating systems and architectures. Nonetheless, the consistency of emulators with the real devices, remains uncertain. To address this issue, our objective is to automatically identify inconsistent instructions that exhibit different behavior between emulators and real devices across distinct privileges, including user-level and system-level privilege.**

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 generated 2,774,649 representative instruction streams and developed a differential testing engine, EXAMINER PRO. With this engine, we compared the behavior of real Arm devices across different instruction sets (A32, A64, T16, and T32) with the popular QEMU emulator, both at the user-level and system-level. To demonstrate the generalizability of EXAMINER PRO, we also tested two other emulators, namely Unicorn and Angr. We find that undefined implementation in Arm manual and bugs of emulators are the major causes of inconsistencies. Furthermore, we discover 17 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.

***Index Terms*—Emulator, Differential Testing, Inconsistent Instructions**

I. 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 [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17] are built based on the state-of-the-art CPU emulators (e.g., QEMU [18], Unicorn [19], Angr [20]) 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 [21], Unicornfuzz [22], FirmAFL [23], P2IM [24], HALucinator [25], and TriforceAFL [26].

The widespread use of software emulation in dynamic analysis is based on the implicit assumption that emulators implement the emulated processor ISA correctly. Particularly in malware analysis, there may be a stronger implicit assumption that the execution results of an instruction on the CPU emulator are identical to those on a real device. However, the validity of these assumptions in reality is uncertain. In fact, implementing CPU emulators correctly is a challenging task, and the execution result can differ due to bugs in the CPU emulator or differences in implementation strategies compared to the real device, as our work has demonstrated. These discrepancies can compromise the reliability of emulator-based dynamic analysis. For example, malware can exploit these differences to evade analysis in the emulator and protect its malicious behavior [27], [28], [29], [30].

In this work, we aim to test Arm emulators across different privileges (i.e., user-level and system-level). Specifically, we will 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. The first challenge is the diversity of the Arm architecture, which includes multiple versions (e.g., ARMv5, ARMv6, ARMv7, and ARMv8), different register widths (32 bits or 64 bits), and various instruction sets (Arm, Thumb-1, and Thumb-2). We need to generate a sufficient number of instruction streams that cover these variants while minimizing the time cost. Simply enumerating all 32-bit instruction streams would be inefficient, and randomly generated streams would not be representative (Section IV-A). The second challenge is to ensure a deterministic execution environment for each test case. We must set up the same CPU state and disable all asynchronous events before executing the instruction stream.

M. Jiang, X. Zheng, R. Chang, Y. Zhou and are with the Zhejiang University, Hangzhou 310027, China. E-mail: jiangmuhui@gmail.com, {xiaoyez, crix1021, yajin_zhou}@zju.edu.cn.

X. Luo is with the Hong Kong polytechnic university, Hong Kong SAR, China. E-mail: csxluo@comp.polyu.edu.hk.

Corresponding author: Rui Chang.

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

After execution, we automatically compare the results to determine any discrepancies.

Locating inconsistent instructions at the system-level poses significant challenges. First, we must address the issue of inspecting the system's state at any time, even in the presence of fatal exceptions. This requires complete control of the system, enabling us to freeze the system and dump its state before and after executing the tested instruction streams. Second, we must effectively handle instructions that may cause the tested machines to enter unusable states, such as kernel panics. These situations require special attention as they can render the system unresponsive and require manual intervention to regain control. Addressing these challenges is essential to achieving reliable and effective system-level testing.

Previous works [31], [32], [33], [34], 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 [33]) 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 [35]) 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 [17], [22], [36], [37]. We will discuss the major differences between iDEV and our work in Section V.

Our system is able to automatically locate inconsistent instructions in a systematic mechanism with the following two key insights.

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) [38] . 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 predefined 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 II-B). 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. At the user-level, the testing process incorporates prologue instructions that initialize the values of general-purpose registers. These instructions set up the execution environment before the instruction stream is executed. The epilogue instructions are responsible for saving the register values onto the stack and writing them into a file. This allows for later comparison to determine if the tested instruction stream exhibits inconsistency. At the system-level, we employ a master-slave architecture to facilitate the testing process and capture the results. The slave machine executes the instruction streams within a kernel module, while the master machine operates the kgdb debugger to capture the system state. The prologue instructions initialize the system state before executing instruction streams. The epilogue instructions delegate system control to kgdb after executing instruction streams, enabling the capture of the system state for subsequent analysis and comparison. This approach allows us to effectively test and analyze the system-level behavior of the instruction streams.

We implement a prototype system called EXAMINER PRO, 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. In user-level, by comparing the result between the state-of-the-art emulator (i.e., QEMU [18]) 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 further verify the capability of EXAMINER PRO in locating inconsistent instructions at the system-level, we execute the test cases within the kernel module and compare the results obtained from QEMU and real devices using different instruction sets (T16, T32, A32, and A64). EXAMINER PRO successfully identifies $60,630$ inconsistent instruction streams, covering 33.0% of the instruction encodings. To demonstrate the generalization of EXAMINER PRO, we extend its application to two additional CPU emulators(i.e., Unicorn [19] and Angr [20]) 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 15 bugs (7 in QEMU [39], [40], [41], [42], [43], [44], [45], 3 in Unicorn [46], 5 in Angr [47], [48], [49], [50], [51]) in emulators and all of them have been confirmed by developers. Furthermore, 2 bugs in real devices (Raspberry Pi 2B and Raspberry Pi 4B) are also explored. 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. In addition, when conducting fuzzing within an emulator, instrumenting the program with inconsistent instructions can significantly reduce the coverage ratio of the fuzzing process, which further reduce the attack surfaces. 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 extend our prototype system, named EXAMINER [52], to EXAMINER PRO. EXAMINER PRO comprises a test case generator and a differential testing engine at both user and system levels. Our experiments showed that EXAMINER PRO is general and can automatically locate inconsistent instructions across different privileges (i.e., user-level and system-level).

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 at both user and system levels. 17 bugs are discovered while 15 of them have been confirmed. 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.

II. BACKGROUND AND MOTIVATION

A. 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.

The emulators usually work in two different privileges, **user-level** and **system-level**. At the user-level, emulators are capable of launching processes that are compiled for one CPU on another CPU. The host machine provides an operating system, allowing for the use of system calls and POSIX

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.

Fig. 1: A motivation example.

signals. Unicorn, which is based on QEMU, provides friendly APIs for building different tools. It aims to emulate CPU operations only and remove other supports, such as signals, to keep it lightweight. Other binary frameworks, such as Angr, also support CPU emulation where users can specify the entry address and execute the target instructions step by step.

At the system-level, emulators provide a virtual model of an entire machine, including the CPU, memory, and I/O peripherals, to run a guest OS. The CPU can be fully emulated or work with a hypervisor such as KVM or Xen. The latter is known as CPU virtualization, which executes most of the guest code directly on the host CPU and only emulates the instructions that cannot be executed natively on the host. Due to the fact that most instructions are run directly on the host, CPU virtualization is out of the scope of this paper.

B. Motivation

EXAMINER PRO can be used to find inconsistent instructions in both user-level and system-level. The inconsistent instructions can be automatically used in many scenarios (Section IV-F). We illustrate how EXAMINER PRO can detect inconsistent instructions and identify bugs in emulators through a motivation example, and highlight the necessity of extending the differential testing engine to system-level with an inconsistent example detected at system-level.

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 [11:11]. The values in other offsets include 6 encoding symbols and they are `Rn`, `Rt`, `P`, `U`, `W`, and `Imm8`.

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 [53].

- 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) Test Case Generation: By analyzing the encoding schema, EXAMINER PRO 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 conditions 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 the negations of the constraints. During this process, we generated 576 instruction streams as test cases in total.

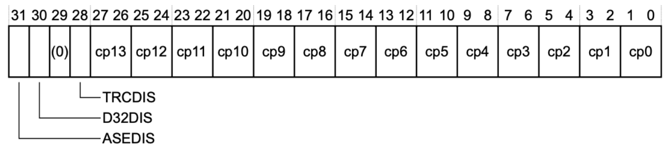
3) Differential Testing: We feed each instruction stream into our differential testing engine (Section III-C), which adds prologue and epilogue instructions. The prologue instructions first set the initial execution context before executing the instruction stream. After the instruction stream is executed, 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 [40]. 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).

4) System-level Extension: System-level emulators are wildly employed in different applications such as hardware simulation and modeling [54], [55], and dynamic malware analysis systems [56], [4]. Therefore, it is crucial to thoroughly test them. Meanwhile, the original EXAMINER cannot test the instructions in privilege mode (system-level). In this case, a system-level extension is necessary. Note that the EXAMINER cannot be directly applied to the system-level due to several challenges encountered (Section III-C3). Thus, we propose EXAMINER PRO that supports locating inconsistent instructions at both user-level and system-level and illustrate a practical example of inconsistent instructions, based on a real defect detected by EXAMINER PRO.

According to the ARMv7-A reference manual [53], advanced SIMD instructions accessed from system-level are undefined instructions when the ASEDIS bit in the CPACR system control register is set to 1, as represented in line 16 of Fig. 3b. We configure the kernel with the ASEDIS equal to 1 and tested SIMD instructions such as `VMAX` and `VADD` on both emulators and real devices at the system-level. The QEMU system-level emulator, i.e., qemu-system-arm, does not completely adhere to the specification and not honour CPACR.ASEDIS, executing the instructions normally.

```
 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    }
```

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



(a) The bit assignments in CPACR system control register

```
 1  CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check,
         boolean advsimd)
 2  // In Non-secure state, Non-secure view of CPACR and
         HCPTR determines behavior
 3  cpacr_cp10 = CPACR.cp10;
 4  cpacr_cp11 = CPACR.cp11;
 5  cpacr_asedis = CPACR.ASEDIS;
 6  ...
 7  if !HaveVirtExt() || !CurrentModeIsHyp() then
 8      if cpacr_cp10 != cpacr_cp11 then UNPREDICTABLE;
 9      case cpacr_cp10 of
10          when '00' UNDEFINED;
11          when '01' if !CurrentModeIsNotUser() then
                UNDEFINED;
12          when '10' UNPREDICTABLE;
13          when '11' // CPACR permits access;
14      // If the Advanced SIMD extension is specified,
15      // check whether it is disabled.
16      if advsimd && cpacr_asedis == '1' then UNDEFINED;
17  ...
```

(b) Related Pseudocode details of enabling the Advanced SIMD Extensions.

Fig. 3: A practical example at system-level.

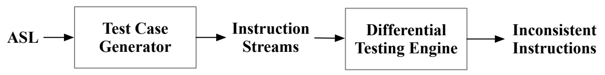


Fig. 4: The work flow of our system

In contrast, the real device was trapped in the undefined exception handler. The incomplete implementation in the emulator resulted in inconsistency between real devices and system-level emulations.

III. Design and Implementation

Figure 4 shows the workflow of Examiner Pro, 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 III-B). Then, the differential testing engine receives the generated test cases and conducts differential testing between the emulators and real devices (Section III-C). 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.

A. Challenges

Locating the inconsistent instructions at both user-level and system-level is not easy.

First, Arm architecture has multiple versions (e.g., ARMv5, ARMv6, ARMv7, and ARMv8), different register widths (32 bits or 64 bits) and instruction sets. Besides, it has mixed instruction modes (Arm, Thumb-1, and Thumb-2). Thus, how to generate sufficient *instruction streams*, which denote the bytecode of an instruction, to cover these variants 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 IV-A). *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 and disable all the effect of asynchronous events before the execution and compare the state afterwards.

To extend the testing to system-level and ensure the capability of Examiner Pro, we have the following two more challenges. *Third*, a new architecture is needed to maintain complete control over the tested machine at system-level, allowing for inspection of its state at the system-level at any time, even in the event of fatal exceptions. We adapted the master-slave architecture proposed by previous work [57] to Arm architecture and set the tested machines in kernel debugging mode so that they can be completely frozen when necessary. *Last but not least*, some testing instructions may lead the tested machines into unusable states (e.g., kernel panic) at system-level, from which it would be impossible to regain the control without a reboot. Rebooting is time-consuming and requires manual intervention when system crashes. Thus, a customized kernel with modified exception handler routine is needed. The customized kernel is supposed to catch the triggered exception and enable the execution of the next test case instead of terminating the testing process.

B. 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 I, 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. Note that we generate test cases based on Arm specifications, covering both privileged and unprivileged instructions. Thus, the generated test cases are suitable for both user-level and system-level.

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 III-B1. 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 III-B2. 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).

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.

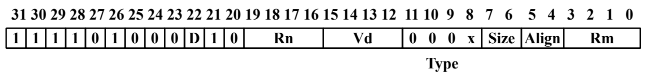
Table I shows the rules to initialize the mutation set. For a register index, we include the `PC` register (index 15), `R0`, `R1`,

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: $T$;
Function Generate($I\_Encode, I\_Decode, I\_Execute$):
    $Symbols$, $Constants$, $Constraints$ = ParseASL($I\_Encode$,
      $I\_Decode$, $I\_Execute$)
    for $S$ in $Symbols$ do
        $S.MutationSet$ = InitSet($S$)
    for $C$ in $Constants$ do
        $C.MutationSet$ = [ConstantValue]
    for $C$ in $Constraints$ + $NegatedConstraints$ do
        ValueSet = SolveConstraint($C$, $Symbols$, $I\_Decode$,
          $I\_Execute$)
        for $V, S$ in $ValueSet$ do
            if $V$ not in $S.MutationSet$ then
                $S.MutationSet$ add $V$
    $MutationSets$ = [$S.MutationSet$ +$C.MutationSet$]
    $TestCase$ = CartesianProduct($MutationSets$)
    return $T$
```

TABLE I: 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

Fig. 5: Test case generator example.

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. Note that we do not mutate 5-bits general purpose registers in AArch64. This decision is based on the fact that there is no such register like PC that can change the execution flow in AArch64. Meanwhile, the eight registers from `X0` to `X7` can all be used to represent return values in AArch64. Mutating these registers may result in redundant test cases. The results of covered constraints and QEMU source code coverage in Section IV-A demonstrate that this choice is adequate. 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.

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 5 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.

More formally, given the ASL program P with a set of statements S and the encoding symbols V, we create the symbolic state $S0$ with symbolic values for the encoding symbols. For each statement S_i in S, we update the $S0$ by symbolically executing S_i in reverse and generating constraints on the encoding symbols V based on the effect of S_i. When there are no further constraints to be generated in the reverse direction, we solve the constraints and their negations for the encoding symbols V and append them to their mutation set accordingly.

Note that path explosion, a common issue in traditional programming languages during symbolic execution, is not a concern in ASL. This is attributed to three key factors: (1) The source code of ASL tends to be relatively simple. (2) The decoding and execution of ASL code impose limited constraints, consequently leading to a restriction in the number of paths. (3) ASL code features a limited number of function calls. However, most of them contribute little to the choice of instruction stream generation. In this case, we model these utility functions (e.g., UInt) to prevent the propagation of symbols into these functions.

Our experiment in Section IV-A shows that we can generate all the test cases within 4 minutes.

C. Differential Testing Engine

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.

In user-level testing, 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 stream I may trigger. If no signal or exception is triggered, the value of Sig is 0.

At the system-level testing, we model the CPU states with user registers, system status, memory, and exceptions. Similarly, we denote CPU T's initial state in system-level $CPU_I^S(T)$ with the tuple $< PC_T, Reg_T, Mem_T, Sta_T^S >$. Inside $CPU_I^S(T)$, all the other attributes have the same meanings as they are inside $CPU_I(T)$ except Reg_T and Sta_T^S. Reg_T excludes SP due to the fact that test cases at system-level may use a different value of SP, which is managed by the kernel and may be inconsistent between emulators and real devices. Sta_T^S denotes the status register in system mode, which is $CPSR$ in Arm architecture. We ignore the other system registers, which are supposed to be consistent after the system initialization. We denote the CPU T's final state in system level $CPU_F^S(T)$ with the tuple $[PC_T, Reg_T, Mem_T, Sta_T^S, Except_T]$. Inside $CPU_F^S(T)$, all the other attributes have the same meanings as they are inside $CPU_I^S(T)$ except $Except_T$. $Except_T$ denotes the exception that the instruction stream I may trigger in system mode, which is *undefined instruction*, *data abortion*, *prefectch abortion* and *supervisor call* in Arm architecture. If no signal or exception is triggered, the value of $Except_T$ is 0. Note that IRQ and FIQ interrupts are disabled to guarantee deterministic execution. Thus, no IRQ nor FIQ interrupts would be triggered during testing.

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:

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

We can identify inconsistent instruction streams at the system-level in a similar way by verifying whether the final state of the CPU emulator, $CPU_F^S(E)$, equals the final state of the real device, $CPU_F^S(R)$, when their initial states $CPU_I^S(E)$ and $CPU_I^S(R)$ are the same.

2) User-level Strategy: To conduct the differential testing at user-level, 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 [58] 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) System-level Strategy: As mentioned before, freezing the system state and recovering from fatal exceptions in system-level can be challenging. We adapted the master-slave architecture proposed by previous work [57] to Arm architecture and customized the kernel to test instructions in system-level. Specifically, the differential testing engine in EXAMINER PRO at system-level is composed of the following modules as shown in Fig. 6: (1) a master machine with gdb to schedule test case execution, freeze and capture system states on the tested machines (slaves); (2) tested machines with customized kernels and kgdb to execute the test cases in kernel module; (3) serial ports that connect the slaves to the master.

Customized kernel. We used the Linux kernel and customized the kernel in two steps. First, we modified the

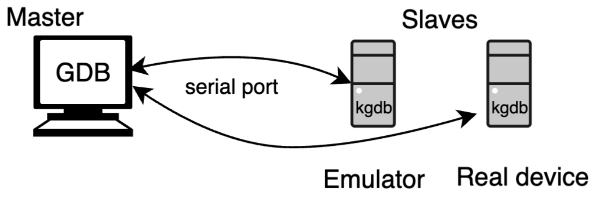


Fig. 6: The architecture of differential testing engine at system-level

exception handlers in the kernel. Specifically, we disabled the kernel panic function in the handlers, which causes the system to abort, and updated the program counter of the occurred exception to the next instruction. Thus, if a fatal exception is triggered, the testing process would be trapped to the specific exception handler, return from the handler, and execute the next instruction. Second, we enable kernel debugging in the kernel configuration and compile the kernel with the `-mfloat-abi=softfp` compilation flag to enable the floating point instructions. In this way, we could freeze the system state through kgdb and test floating instructions.

Test case template. For each test case, we generated a kernel module template with prologue and epilogue instructions inserted around the testing instruction stream, similar to the user-level strategy. To ensure the initial state consistency in system-level, we set the value of all general purpose registers to zero, except PC as user-level, and set the mask bits in the status register CPSR to disable asynchronous exceptions. After initialization, we insert a kgdb breakpoint to dump the CPU initialization state at system-level. Then we proceed to test the actual test instruction stream and insert another kgdb breakpoint to dump the CPU final state. We also set breakpoints in exception handlers to intercept the exception triggered by the tested instruction. For efficiency, we compiled every 100,000 test instructions into one kernel module and reuse the prologue and epilogue instructions.

Testing flow. The testing flow of the differential engine in system-level is illustrated in Fig. 7. The master maintains a debugger, i.e., gdb, that issues commands to and transfers data back from the slaves. ①At the beginning, the slave boots the customized system and signals its readiness to the master. ②Then the master sets up the dump directory and sets the breakpoints to dump the system state when testing. ③Next, the master releases the slave, which then loads the kernel module. ④Finally, the tested instruction streams in the kernel module are tested one by one in terms of rounds. In each round, the master interacts with the slave in three main phases. First, the master sets PC to the current testing instruction by calculating the instruction offset. Second, the master releases the slave, which then executes the test case's initialization code and is intercepted by the first breakpoint. Third, the slave executes the actual instruction being tested and is intercepted by the second breakpoint. The master dumps this final state, either when the instruction finishes execution or at exception handlers, and uses it to determine whether the tested instruction streams are inconsistent between emulators and the real device.

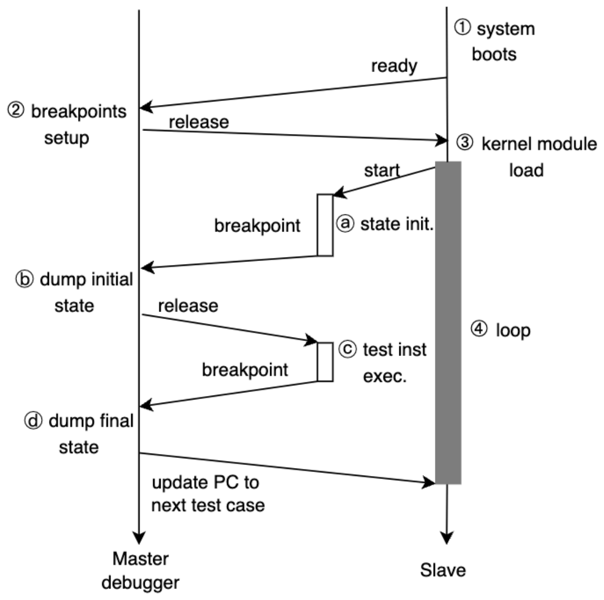


Fig. 7: The testing flow at system-level

D. Implementation Details

We implement EXAMINER PRO 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 [59] 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 is implemented with inline assembly code. The execution result dumping is implemented with inline assembly code at the user-level and with the use of gdb scripts at the system-level. In total, EXAMINER PRO contains $5,074$ lines of Python code, 220 lines of C code, and 200 lines of assembly code and adds another 723 lines of Python code, 277 lines of C code with inline assembly, and 351 lines of gdb scripts for extending to system-level testing.

IV. EVALUATION

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

- **RQ1:** Is EXAMINER PRO able to generate sufficient test cases?
- **RQ2:** Is EXAMINER PRO able to detect inconsistent instructions at user-level? What are the root causes of these inconsistent instructions?
- **RQ3:** Is EXAMINER PRO able to detect inconsistent instructions at system-level? What are the root causes of these inconsistent instructions?
- **RQ4:** What is the difference between inconsistent instructions detected by EXAMINER PRO at user-level and system-level?
- **RQ5:** Is EXAMINER PRO general to be applied to the other emulators?

TABLE II: The statistics of the syntactically correct generated instruction streams. "EXAMINER PRO " denotes the number of generated test cases by our test case generator. "Random" denotes the number of randomly generated test cases. "Symbol Random" denotes the test case generator with symbolic execution turned off and a few additional random values chosen for symbols. 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 EXANINER Pro, Random, / Symbol Random	Instruction Stream			Instruction Encoding			Instruction			Covered Constraints		
		EXAMINER Pro	Random	Symbol Random	EXAMINER Pro	Random	Symbol Random	EXAMINER Pro	Random	Symbol Random	EXAMINER Pro	Random	Symbol Random
A64	70.51, 56.18, 1.54	1,041,355	421,645	573,977	837	266	839	579	178	581	3,430	935	3,061
A32	75.05, 40.43, 1.03	949,704	582,849	641,995	550	415	550	481	360	481	5,318	3,716	4,935
T32	74.58, 37.90, 0.89	873,921	34,598	478,143	530	351	531	450	283	451	5,008	3,203	4,675
T16	2.32, 0.044, 0.023	928	796	890	77	57	78	67	49	68	120	84	107
Overall	222.46, 134.55, 3.48	2,865,910	1,039,890	1,695,006	1,994	1,088	1998	1,070	550	1,070	13,875	7,938	12,778

- **RQ6:** What are the possible usage scenarios of inconsistent instructions?

A. *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 cases is sufficient. Table II shows the statistics of the generated instruction streams. Note that we repeat the generation process 10 times and calculate the average value to eliminate randomness. In total, $2,865,910$ instruction streams are generated within 4 minutes, which cover $1,994$ 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. In addition, the total number of constraints (including their negation) for A64, A32, T32, and T16 is $3,690$, $5,838$, $5,682$, and 174, respectively. This shows that our test cases can cover 93.0%, 91.1%, 88.1%, and 69.0% of the constraints, respectively. 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 13 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 compare it with two different strategies: (1) randomly generating test cases for each instruction set (*random* strategy); (2) using a test case generator with symbolic execution turned off and a few additional random values chosen for the symbols (*symbol random* strategy). The *random* strategy compares the performance of our generator with those used in existing literature [33], [34]. The *symbol random* strategy provides insight into the significance of symbolic execution. We generate test cases of the same order of magnitude for all three strategies to ensure a fair comparison. Specifically, we produce the same number of test cases as our test case generator for the other two strategies. We do not allocate the same time budget here, as the number of instructions generated by these two strategies would substantially increase, resulting in time-consuming testing for a potential slight performance improvement. Additionally, all strategies are capable of generating test cases for four instruction sets within five minutes. Note that randomly generated instructions may not be syntactically correct, leading to variance in the number of valid instructions. To eliminate the effects of randomization in these two strategies, we also repeat the generation process for 10 times and report the average value in Table II. We calculate how many instruction encodings, how many instructions, and how many constraints are covered by these instruction streams for these two strategies. According to the Column "Random" in Table II, only 36.3% ($\frac{1,039,890}{2,865,910}$) 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. The Column "Symbol Random" in Table II shows that, although *symbol random* can encompass all instruction encodings and instructions, it only covers 92% of the constraints that our test case generator can address. We report p-values to show the significant difference between our test case generator and the other two strategies. The p-values between our test case generator and the random strategy are 1.44×10^{-4}, 1.63×10^{-4}, and 1.47×10^{-4} for instructions, instruction encoding, and covered constraints, respectively. The p-values between our test case generator and the symbol-random strategy are 4.77×10^{-5}, 5.59×10^{-5}, and 1.46×10^{-4} for instructions, instruction encoding, and covered constraints, respectively. We also utilize statistical tests, specifically the Mann–Whitney U test with Cliff's delta effect size to analyze and compare our strategy with *random* and *symbol random* strategy. There are statistically significant improvements in instruction encoding, instruction, and covered constraints between the *random* strategy and our test case generator at A64, A32, T32, T16 (Cliff's delta is either 1 or

TABLE III: QEMU code coverage across various strategies. ”User level” denotes the code coverage of qemu-arm and qemu-aarch64. ”System level” denotes the code coverage of qemu-system-arm and qemu-system-aarch64. “Combined Coverage” denotes the combined code coverage of both ”User level” ones and ”System level” ones.

Instruction Set	Test Case Coverage								
	User level			System level			Combined Coverage		
	EXAMINER PRO	Random	Symbol Random	EXAMINER PRO	Random	Symbol Random	EXAMINER PRO	Random	Symbol Random
A64	20.3%	21.0%	19.9%	20.3%	19.6%	19.9%	15.2%	15.8%	14.9%
A32	50.0%	50.9%	48.2%	42.5%	41.2%	40.7%	49.1%	50.7%	47.5%
T32	57.2%	50.7%	54.4%	57.5%	47.3%	49.7%	59.3%	53.4%	56.6%
T16	13.5%	12.9%	13.5%	22.6%	22.2%	23.6%	20.1%	19.6%	20.8%
Overall	42.1%	41.4%	41.6%	42.1%	39.3%	40.2%	38.4%	37.9%	38.2%

-1). The *symbol random* strategy and our test case generator also show a statistical difference in instruction encoding, instruction, and constraints at A64, T32, T16 ((Cliff's delta is either 1 or -1)).

To substantiate the claim of test sufficiency, we measure code coverage of QEMU's source code using test cases generated by our symbolic execution at both system and user level. We subsequently compare this coverage with that achieved by randomly generated test cases and when symbolic execution is disabled. Note that we filter out irrelevant tracing code in QEMU and calculate the coverage only for QEMU source code related to instruction decoding and translation. As shown in Table III, the coverage of EXAMINER PRO is slightly higher than that of the *random* and *symbol random* strategies. This suggests that our test cases could cover more situations. One possible explanation for the modest increase in coverage is that most of the QEMU code is easy to access, and rare unhandled situations only account for a limited portion. Additionally, QEMU's implementation may lack certain logic to differentiate subtle variations in the instruction streams. For example, QEMU does not support `LDC` instructions, and all these instructions would raise an undefined exception in QEMU, which only cover the undefined source code. The subtle variations in the `LDC` would not increase the coverage here. Note that the code coverage of QEMU is not particularly high because there is always another mode of code, whether in user mode or system mode, that cannot be tested. The overall coverage ratio may appear lower when combining the coverage of user mode and system mode due to the fact that they can cover the same code, resulting in an increase in the total relevant code being considered during the calculation.

Answer to RQ1: EXAMINER PRO can generate sufficient test cases, which are all syntactical correct instruction streams and can cover all instruction encodings and instructions. On the contrary, 45.5% instruction encodings, 48.6% instructions, and 37.4% constraints cannot be explored by these randomly generated instructions. With symbolic execution turned off, the generated instruction streams only cover 92% of constraints. Furthermore, the test cases generated by our test case generator cover a greater portion of QEMU source code at both the user and system levels.

```
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
```

Fig. 8: Two different implementations are defined in the annotation of function ExclusiveMonitorsPass, which is called by many instructions' executing code

B. *Differential Testing Results at User-level* (***RQ2***)

We feed the generated test cases into our differential testing engine to locate the inconsistent instructions at user-level. Table IV 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. Thanks to the representative test cases, the differential testing for all the test cases can be finished within 13 hours. 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 5,276 seconds to 46,238 seconds (13 hours), depending on the specific devices. Meanwhile, iDEV [35] generates approximately 33 million test instruction streams, which is ten times more than our generated test cases. If we were to utilize those test cases, the waiting time for results would be approximately 130 hours (roughly five days) for the most time-consuming testing scenario. This further demonstrate that our required time is acceptable.

Testing Result According to Table IV, 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.

TABLE IV: The results of differential testing for QEMU at user-level. "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

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 [39], [40], [41], [42] in total, which influence 11 instruction encodings. Some of the bugs are related to very common instructions. The first bug influences the `BLX` instruction [39]. 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 [40] 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 [41] (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 [42] 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 PRO 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 II-B). 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 signifies behavior defined by the implementation, necessitating vendors to define and document how their implementation behaves. Figure 8 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. 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 PRO 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 PRO 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`).

C. Differential Testing Results at System-level (*RQ3*)

We feed the generated test cases into our differential testing engine at the system-level to locate inconsistent instructions. The results are shown in Table V.

Experiment Setup. We conduct the differential testing between QEMU (version 7.2.0) at system-level and two real devices (RaspberryPi 2B in ARMv7, and RaspberryPi 4B in ARMv8). ARMv7 is backward compatible with ARMv5 and ARMv6, with the main difference lying in processor extensions. To optimize testing time, we focused our efforts on the A32, T32, and T16 instruction sets on ARMv7 at the system level.

The testing process for each instruction set on QEMU, running on the Intel(R) Core(TM) i7-8700 CPU, took approximately 213s to 8,992s of CPU time and 3.4 hours to 90 hours of total testing time. On the other hand, testing 1,000 instruction streams on the real devices took around 13 minutes to 30 minutes. The total time required for the entire testing process ranged from approximately 7 to 22 days, depending on the specific devices. To expedite the process, we ran multiple boards of the same real devices in parallel. Note that the total testing time was significantly longer than the CPU time due to the additional time taken by the serial port debugging.

Testing Result. According to the data presented in Table V, we discovered a total of $60,630$ inconsistent instruction streams, which account for 2.3% of all the test cases conducted. These inconsistent instruction streams cover 606 different instruction encodings and 278 instructions, owning 33.0% and 31.7% of the tested instruction encodings and instructions, respectively. Note that branch instructions (e.g. `B`) or arithmetic instructions which update PC or SP, would lead the testing system to an unrecoverable state or crash. Thus, we filter out these instructions in the experiment.

Inconsistent Behaviors. In ARMv7, most of the inconsistent instruction streams (i.e., 84.5% and 74.5%) will trigger different exceptions between the real device and emulators. In a smaller portion of cases (i.e., 15.4% and 25.3%), the instruction streams may not trigger an exception but still exhibit differences in register or memory values. In ARMv8, most of the inconsistent streams (i.e., 91.2%) will have different register or memory values. Specifically, the inconsistency is caused by the difference in CPSR register. Similar to RQ1, we categorize instructions that cause QEMU to crash but are executed normally on real devices, or vice versa, as ”Others.”

Root Cause. We analyze the root cause of these inconsistent instructions. First, there are implementation bugs in QEMU or real devices, whose behaviour is not cohere to Arm specification. We found out 3 bugs in QEMU [43], [44], [45], 1 bug in Raspberry Pi 2B, and 1 bug in Raspberry Pi 4B, which influence 133 instruction encodings. The first bug we identified in QEMU [43] is related to the unimplemented LDC instructions. QEMU does not support the Debug Communications Channel which is specified in the Arm architecture. As a result, when executing LDC instructions, QEMU raises an undefined exception. In contrast, these instructions execute successfully on the real device without triggering any exceptions. The second bug in QEMU [44] influences many SIMD instructions (e.g., VMAX and VADD) in specific system settings. QEMU at the system-level does not honor the ASEDIS bit in the CPACR register. When the system disables the SIMD instructions, QEMU still can execute these instructions successfully, leading to inconsistent behavior between QEMU and the real devices. The last bug we identified in QEMU [45] is related to the configuration of two system registers, namely `CCSIDR` and `DBGDRAR`. These registers are not properly matched in the emulated real devices, resulting in register inconsistency between QEMU and the real devices.

The bug we identified in Raspberry Pi 2B affects the PLD instruction. According to the ARMv7 manual, the PLD instruction is a memory system hint and can be treated as a NOP without affecting the functional behavior of the device. However, Raspberry Pi 2B triggers an undefined exception when executing the instruction. The bug we identified in Raspberry Pi 4B affects certain cryptographic instructions, such as AESD (AES decryption) and SHA256H2 (SHA-256 hashing). These instructions should be implemented with the cryptography extension enabled. However, in the case of Raspberry Pi 4B, executing these instructions triggers undefined exceptions. Similar to QEMU at user-level, most of the inconsistent instructions are due to the unpredictable implementation in the Arm manual in ARMv7. Additionally, at the system-level, the test cases may use different initial values of the stack pointer, which leads to inconsistencies in the final state. Therefore, we classify inconsistent instructions that read the value of SP as ”Others”.

Answer to RQ3: EXAMINER PRO can detect inconsistent instructions. In total, 60,630 inconsistent instruction streams are found, which covers 33.3% (i.e., 606/1,834) instruction encodings and 31.7% instructions (i.e., 278/877).

TABLE V: The results of differential testing for QEMU at system-level. The attributes denotes the same meaning explained in the caption of Table IV.

Architecture	ARMv7		ARMv8	Overall
Experiment Setup				
Instruction Set	A32	T32&T16	A64	-
QEMU Binary	qemu-system-arm		qemu-system-aarch64	-
QEMU Model	Cortex-A7		Cortex-A72	-
Device Name	RaspberryPi 2B		RaspberryPi 4B	-
CPU Time / 1,000 Inst_S (Device)	1.02s	0.38s	2.19s	-
Total Time / 1,000 Inst_S (Device)	1061.65s	782.39s	1,780.95s	-
CPU Time / 1,000 Inst_S (QEMU)	3.28s	0.28s	8.29s	-
Total Time / 1,000 Inst_S (QEMU)	15.05s	429.05s	43.11s	-
Tested Inst_S	801,583	761,190	1,084,692	2,647,465
Tested Inst_E	476	529	829	1,834
Tested Inst	299	301	571	877
Testing Result	The percentage is based on the number of inconsistent instructions (streams/encodings)			
Inconsistent Inst_S	19,805 | 2.5%	20,164 | 2.6%	20,661 | 1.9%	60,630 | 2.3%
Inconsistent Inst_E	323 | 67.9%	219 | 41.4%	64 | 7.7%	606 | 33.0%
Inconsistent Inst	211 | 70.6%	153 | 50.8%	54 | 9.5%	278 | 31.7%
Inconsistent Behaviors	The percentage is based on the number of inconsistent instructions (streams/encodings)			
Exception (Inst_S)	16,742 | 84.5%	15,032 | 74.5%	1,805 | 8.7%	33,579 | 55.4%
Exception (Inst_E)	280	170	7	457
Exception (Inst)	210	145	7	227
Register/Memory (Inst_S)	3,041 | 15.4%	5,095 | 25.3%	18,839 | 91.2%	26,975 | 44.5%
Register/Memory (Inst_E)	148	168	55	371
Register/Memory (Inst)	100	110	45	170
Others (Inst_S)	22 | 0.1%	37 | 0.2%	17 | 0.1%	76 | 0.1%
Others (Inst_E)	8	13	4	25
Others (Inst)	8	12	4	22
Root Cause	The percentage is based on the number of inconsistent instructions (streams/encodings)			
Bugs (Inst_S)	477 | 2.4%	198 | 1.0%	425 | 2.1%	1,100 | 1.8%
Bugs (Inst_E)	125	3	5	133
Bugs (Inst)	72	3	5	78
UNPRE. (Inst_S)	16,303| 82.3%	15,320 | 76.0%	1,454 | 7.0%	33,077 | 54.6%
UNPRE. (Inst_E)	168	169	7	344
UNPRE. (Inst)	149	144	7	176
Others (Inst_S)	3,025 | 15.3%	4,646 | 23.0%	18,782 | 90.9%	26,453 | 43.6%
Others (Inst_E)	147	164	54	361
Others (Inst)	99	106	44	168

D. *Difference between Inconsistent Instructions* **(RQ4)**

Testing Result. We discovered that the system-level testing in the ARMv7 architecture revealed fewer inconsistent instruction streams compared to the user-level testing. One possible reason for this difference could be that we tested different versions of QEMU at the user-level (version 5.1.0) and system-level (version 7.2.0). Certain bugs or inconsistencies in the earlier version of QEMU used in the user-level testing were fixed in the later version used in the system-level testing. In the ARMv8 architecture, the system-level testing revealed more inconsistent instruction encodings and instructions compared to the user-level testing. This difference could be attributed to the different initial state of the Stack Pointer (SP) at system-level as mentioned in RQ3. The variation in SP's initial state could lead to different behaviors and outcomes for certain instructions, resulting in a higher number of inconsistencies observed at the system level.

Inconsistent Behaviors. In ARMv7, we observed that both system-level and user-level testing approaches exhibited similar inconsistent behaviors. Most of these inconsistencies were caused by differences in signals or exceptions triggered by the tested instructions. This implies that the discrepancies between the emulated environment and the real device primarily manifest as variations in the exception handling mechanisms. However, in the ARMv8 architecture, we found that system-level testing yielded more prominent inconsistent behaviors related to different register or memory values.

Root Cause. In ARMv7, we observed that a significant portion of the inconsistent instructions at both user-level and system-level testing can be attributed to the unpredictable implementation described in the Arm manual. In ARMv8, we observed that the different initial system state became a major factor contributing to the inconsistencies at the system-level. When it comes to user-level testing, the buggy inconsistent instructions are primarily caused by bugs or misbehavior of specific instructions in the QEMU emulator. On the other hand, at the system-level testing , the buggy inconsistencies are more closely related to the system status and system setup. These discrepancies can occur due to differences in the initialization process, handling of system registers, or other system-level factors that influence the execution of instructions. Besides, bugs in real devices are discovered at the system-level.

Answer to RQ4: A large number of inconsistent instruction streams were discovered in both user-level and system-level testing. In terms of inconsistent behaviors, both the user-level and system-level testing approaches in ARMv7 exhibited similar patterns and root causes. However, in ARMv8, the system-level testing revealed more significant inconsistencies, particularly related to variations in register or memory values.

E. Generalization of Examiner Pro *(RQ5)*

To demonstrate the generality of Examiner Pro, we further apply Examiner Pro 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 focus solely on emulating the CPU and do not emulate the entire system. These emulators lack system-level emulation and do not offer options for instruction privilege. Consequently, our testing of these emulators is limited to the user level. At the user level, both 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 [60]. For instance, many SIMD instructions will make Angr crash, resulting in 5 new bugs. Instructions (e.g., `WFE` [61]) 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 VI 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 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 RQ5: Examiner Pro is general to be applied to the other CPU emulators (i.e., Unicorn and Angr). With Examiner Pro, 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).

F. Applications of Inconsistent Instructions ***(RQ6)***

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.

```
 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  }
```

Fig. 9: Pseudo code of the native code for detecting the emulator.

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 9 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 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 VII shows the evaluation result, all the mobile apps can detect the existence of emulators and real mobiles successfully.

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 [62].

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 [63])

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

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 VII: 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	✓	✓	✓

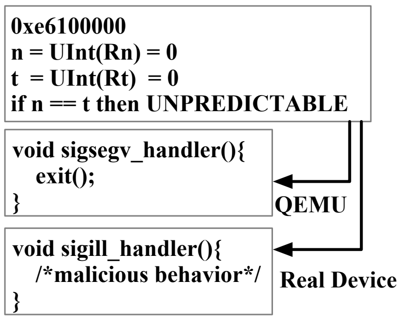


Fig. 10: Inconsistent instruction can prevent the malicious behavior being detected by emulators

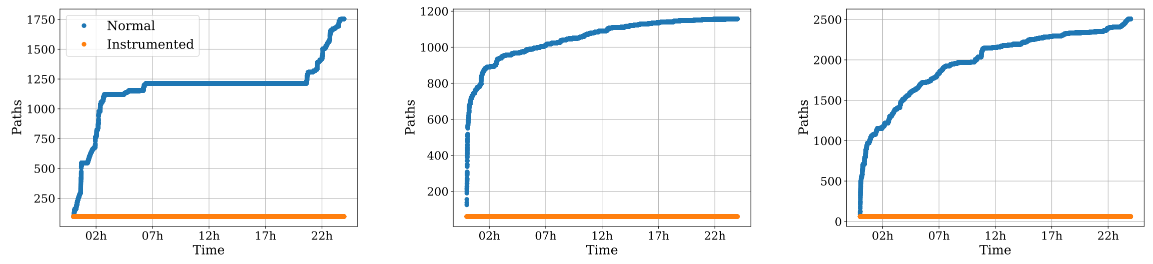
```
0x10000: e51b3008 LDR r3,[fp,#-8]
0x10004: e1a03000 MOV r3,r0
0x10008: e7cf0e9f BFC r0, #0xf, #1
// BFC instruction is to clear specific bits
// e7cf0e9f is an UNPREDICTABLE encoding
// e7cf0e9f is executed normally in real device
// e7cf0e9f triggers SIGILL signal on QEMU
0x1000c: e1a00003 MOV r0,r3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

Fig. 11: Instrumented instruction streams for anti-fuzzing.

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 [64]) to Debian 7.3. As shown in Figure 10, 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*.

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 [65], [66]. Considering that many new binary fuzzing frameworks are based on QEMU [21], [24], [25], [23], 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 11. 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



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

Fig. 12: 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.

TABLE VIII: 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.

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 VIII 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 VIII. We collect the coverage information for the instrumented and the normal ones. Figure 12 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 V).

Answer to RQ6: 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.

V. Discussion

A. Advancement over iDEV [35]

Though both Examiner Pro and iDEV use differential testing with generated test cases, Examiner Pro has better scalability and capability on locating inconsistent instruction streams in terms of the following perspectives. 1) **Test case generation**: Examiner Pro 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 Pro 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 at user-level, 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 Pro 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 Pro 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 Pro. 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.

B. Significance of Signal Inconsistency and Exception Inconsistencies

In Examiner-Pro, the modeled CPU state is different between user-level and system-level, which is discussed in detail in Section III-C. Apart from the program counter, registers, memories, and the status register, user level will check the signal inconsistency while system level will check the exception inconsistency after the tested instruction stream is executed. The potential effects of signal inconsistencies and exception inconsistency are from the following 3 aspects. 1) **System Stability**: signal inconsistencies may affect the specific user applications at the user-level. However, at the system-level, exception inconsistencies can have broader implications. A mis-triggered exception at the system level can result in system crashes, instability, or even system-wide failures. 2) **Security Risks**: signal inconsistencies at the user level may create opportunities for privilege escalation or unauthorized access within the application's context. However, at the system level, exception inconsistencies can be more severe and pose significant security threats. An attacker exploiting exception inconsistencies at the system-level may gain control over critical system resources or disrupt essential operations. 3) **Resource Utilization**‘: In user-level, signal inconsistencies may impact resource management within the application's scope. For example, if a signal meant to release resources is not properly handled, it can result in resource leaks or inefficient resource utilization. At the system-level, exception inconsistencies can have broader resource management implications. Mis-triggered exceptions can lead to resource exhaustion, system-wide performance degradation, or denial-of-service scenarios affecting multiple applications or processes.

C. Threats to Validity

Selection of Testing Real devices and Emulators The selection of different real devices and emulators may yield varying inconsistency results. Potential bugs in the implementation of a real device could result in false negatives during differential testing. Emulators and real devices might adopt different implementation choices for unpredictable instructions, leading to variability in the number of inconsistent instructions. To mitigate this threat, we conducted tests on three emulators, assessing whether the inconsistent results align with the ISA. The testing on Arm's own emulators may help us find real device bugs. However, most of the Arm emulators are closed source and they are not widely used in the different kinds of dynamic analysis tools (e.g., fuzzing, malware analysis, crash analysis). Our focus centers more on dynamic analysis tools such as QEMU and angr, aimed at detecting emulator bugs. Future work could extend the application of EXAMINER PRO to testing on Arm emulators.

Internal Validity We implemented a symbolic execution engine for ASL and all scripts by ourselves with careful review; despite the extensive testing phase, we may not exclude all possible implementation errors. For the sake of replicability, we made all data and scripts employed publicly available [2]. Another aspect of internal validity lies in the errors in the manual analysis phase of differential testing results, as well as the applicability and generality of the inconsistencies analysis. We made efforts to automate the analysis of inconsistencies detected by our differential engine. The majority of inconsistencies caused by unpredictable instructions can be automatically identified. We only manually checked the corner cases, which require careful attention and might potentially reveal emulator bugs.

D. Limitations and Future Works

Testing Instructions in Privileged Environments We successfully extend EXAMINER PRO to system-level and conducted system-level testing of instruction streams. However, during the testing process, we encountered occasional crashes (0.1%-0.2%) in the master-slave architecture, as shown in Table IV. When such crashes occurred, manual intervention was required to restart the system and resume testing. Despite these occasional crashes, the majority of instructions could be automatically tested by EXAMINER PRO. To ensure the stability of the system-level testing, we filtered out instructions that update the Program Counter (PC), such as branch instructions. Randomly updated PC values can lead to unrecoverable system states, preventing us from capturing the system state through kgdb and hindering the automatic testing process. We left a more sophisticated test case design of updated PC instruction as a future work.

Testing Instruction Stream Sequences EXAMINER PRO 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. For example, our testing does not include the IT blocks in the T32 instruction set. 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 PRO, covering 29.5% of instructions. Every instruction stream sequence that contains the inconsistent instruction stream can result in inconsistent behaviors.

Instruction Execution Context The initial state of the differential engine may affect the execution result of instruction. Instruction would explore different ASL decoding paths based on different register values and system status. We initialized all general-purpose registers to zero, potentially resulting in false

[2]https://github.com/ZXXYy/ExaminerPro

negatives and overlooking certain behaviors. For example, the majority of memory access instructions may trigger a data abortion exception since the base register for calculating addresses is set to zero, and we do not map the accessed memory to pages before executing the memory access instruction. Exploring more diverse initial states for a broader range of instruction behaviors poses a challenge, and addressing this is deferred to future work.

Detecting (Ab)Used Inconsistent Instructions Section IV-F 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.

Other Architectures The whole framework of EXAMINER PRO 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.

VI. RELATED WORK

A. Testing CPU Emulators

Several works are proposed to test the CPU emulators across different architectures.

For x86/x64 architecture, Lorenzo et al. proposed EmuFuzzer to test the CPU emulators [33], [34]. However, the seed used for testing mainly relies on randomization and a CPU-assisted mechanism, which may not cover sufficient CPU behaviors. KEmuFuzzer is proposed to test the whole system emulators [32]. However, KEmuFuzzer relies on the manually written template to generate test cases and is tested on a hardware-assisted virtual machine rather than a real physical CPU. Shi et al. [57] proposed a method to test instructions at the system level by carefully crafting test cases based on the instruction semantics defined in Intel's x86 CPU manual. Their approach utilized a master-slave architecture to execute and test the instructions. However, human effort is still required in the process of generating the test cases. PokeEMU [31] 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. We can translate the ASL into a sequential emulator using Sail [67], which supports the automatic generation of emulator code in both C and OCaml. The sequential emulator, conforming to ASL, can be regarded as a high-fidelity emulator, covering a significant portion of ISA specifications. By adopting a similar strategy to PokeEMU, we can automatically explore more initial machine states for tested instructions, leading to a broader coverage of CPU behaviors. We have identified this as a potential avenue for future work.

For Arm architecture, iDEV [35] 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 [17], [22], [36], [37]. Proteus [68] aimed to identify instruction discrepancies between Arm CPUs and Android emulators. However, they only use accurate software models of Arm CPUs instead of real physical CPUs.

B. Verifying Physical CPUs

Several approaches have been proposed to enhance the generation of processor-level test programs for processor verification, which is complementary but related to emulator testing. Model-based test generators use an input format specification to guide the generation process, integrating constraints processed by a CSP/SMT solver [69], [70], [71], [72]. IBM's Genesys-Pro [69] enables users to define any desired verification scenario and a test case template. It generates a test by formulating and solving a separate constraint problem using a CSP solver for each test instruction. Bauereiss et al [72] developed a test generator to verify the intended security property of the Arm Morello architecture. Instead of performing symbolic execution directly on the ASL as our work, they translated ASL into Isabelle/HOL and utilized the Isla symbolic execution tooling for Sail [67] to automatically generate interesting instruction-sequence tests. However, they excluded non-deterministic parts of the specification, such as unpredictable instructions when generating test cases, which should be included in our testing. Katz et al [73] presented an optimized test generation framework that effectively propagates constraints among multiple instructions. Mutation-based fuzzing is also employed in processor verification. DifuzzRTL [74] developed a register-coverage guided fuzzing technique to automatically discover unknown bugs in CPU RTLs.

C. Differential Testing

Differential testing is introduced by McKeeman et al. [75] 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) [76] and many other differential testing tools are built based on EMI to validate the compiler implementations [77], [78]. 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 [79] to test the performance regression bugs in DBMSs . Furthermore, differential testing is powerful and applied to different domains such as testing SMT solvers [80], [81], JVM implementations [82] , symbolic execution engines [82], and PDF readers [83].

D. Anti-Emulation Technique

Previous works [27] 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. [28] 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.

VII. Conclusion

We design and implement EXAMINER PRO, a framework that can automatically locate the inconsistent Arm instructions across different privileges. With EXAMINER PRO, we generate 2,774,649 representative instruction streams and detect $171,858$ and $60,630$ inconsistent ones for QEMU at user-level and system-level respectively. To demonstrate EXAMINER PRO's generalization, we further apply EXAMINER PRO 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 17 bugs (7 in QEMU, 3 in Unicorn, 5 in Angr, 2 in real devices). 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.

References

[1] "64 bit juno r2 arm® development platform," https://developer.arm.com/-/media/Arm%20Developer%20Community/-PDF/Juno%20r2%20datasheet.pdf.

[2] A. Davanian, Z. Qi, Y. Qu, and H. Yin, "Decaf++: Elastic whole-system dynamic taint analysis," in *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses*, 2019.

[3] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "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*, 2014.

[4] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Security Symposium*, 2012.

[5] A. Alwabel, H. Shi, G. Bartlett, and J. Mirkovic, "Safe and automated live malware experimentation on public testbeds," in *Proceedings of the 7th Workshop on Cyber Security Experimentation and Test*, 2014.

[6] J. Wei, L. K. Yan, and M. A. Hakim, "Mose: Live migration based on-the-fly software emulation," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015.

[7] C. Carmony, X. Hu, H. Yin, A. V. Bhaskar, and M. Zhang, "Extract me if you can: Abusing pdf parsers in malware detectors." in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, 2016.

[8] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, 2011.

[9] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection and monitoring through vmm-based "out-of-the-box" semantic view reconstruction," *ACM Transactions on Information and System Security*, 2010.

[10] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, 2016.

[11] Q. Feng, A. Prakash, H. Yin, and Z. Lin, "Mace: High-coverage and robust memory analysis for commodity operating systems," in *Proceedings of the 30th annual computer security applications conference*, 2014.

[12] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu, "Repackage-proofing android apps," in *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[13] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," in *Proceedings of the 2020 Annual Computer Security Applications Conference*, 2020.

[14] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted firmware rehosting for embedded systems," in *Proceedings of the 30th USENIX Security Symposium*, 2021.

[15] Z. L. Chua, Y. Wang, T. Baluta, P. Saxena, Z. Liang, and P. Su, "One engine to serve'em all: Inferring taint rules without architectural semantics." in *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, 2019.

[16] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding linux malware," in *Proceedings of the 2018 IEEE symposium on security and privacy*, 2018.

[17] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. Butler, "Firmusb: Vetting usb device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[18] "Qemu," https://www.qemu.org/.

[19] "Unicorn," https://www.unicorn-engine.org/.

[20] "Angr," https://angr.io/.

[21] "Afl qemu mode: high-performance binary-only instrumentation for afl-fuzz," https://github.com/google/afl/tree/master/qemu_mode.

[22] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," in *Proceedings of the 13th USENIX Workshop on Offensive Technologies*, 2019.

[23] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation," in *Proceedings of the 28th USENIX Security Symposium*, 2019.

[24] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *Proceedings of the 29th USENIX Security Symposium*, 2019.

[25] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "Halucinator: Firmware re-hosting through abstraction layer emulation," in *Proceedings of the 29th USENIX Security Symposium*, 2020.

[26] "TriforceAFL," https://github.com/nccgroup/TriforceAFL.

[27] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators," in *Proceedings of the 2007 International Conference on Information Security*. Springer, 2007.

[28] D. Jang, Y. Jeong, S. Lee, M. Park, K. Kwak, D. Kim, and B. B. Kang, "Rethinking anti-emulation techniques for large-scale software deployment," *Computers & Security*, 2019.

[29] A. Issa, "Anti-virtual machines and emulations," *Journal in Computer Virology*, 2012.

[30] C. V. Liţă, D. Cosovan, and D. Gavriluţ, "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.

[31] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "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*, 2012.

[32] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi, "Testing system virtual machines," in *Proceedings of the 19th international symposium on software testing and analysis*, 2010.

[33] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing cpu emulators," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009.

[34] L. Martignoni, R. Paleari, A. Reina, G. F. Roglia, and D. Bruschi, "A methodology for testing cpu emulators," *ACM Transactions on Software Engineering and Methodology*, 2013.

[35] S. Qin, C. Zhang, K. Chen, and Z. Li, "idev: exploring and exploiting semantic deviations in arm instruction processing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[36] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, "Your exploit is mine: Automatic shellcode transplant for remote exploits," in *Proceedings of the 38th IEEE Symposium on Security and Privacy*, 2017.

[37] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, "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*, 2017.

[38] A. Reid, "Trustworthy specifications of arm® v8-a and v8-m system level architecture," in *Proceedings of the 16th Formal Methods in Computer-Aided Design*, 2016.

[39] "Blx instruction bug in qemu," https://bugs.launchpad.net/qemu/+bug/1925512.

[40] "Str instruction bug in qemu," https://bugs.launchpad.net/qemu/+bug/1922887.

[41] "Unaligned data access bug in qemu," https://bugs.launchpad.net/qemu/+bug/1905356.

[42] "Wfi instruction bug in qemu," https://bugs.launchpad.net/qemu/+bug/1926759.

[43] "Simd bug in qemu system," https://gitlab.com/qemu-project/qemu/-/issues/1500.

[44] "System register configuration bug in qemu system," https://gitlab.com/qemu-project/qemu/-/issues/1499.

[45] "Ldc unimplementation in qemu system," https://gitlab.com/qemu-project/qemu/-/issues/1498.

[46] "Bugs in unicorn," https://github.com/unicorn-engine/unicorn/issues/1424.

[47] "Attributeerror bug in angr," https://github.com/angr/angr/issues/2803.

[48] "Vabs bug in angr," https://github.com/angr/angr/issues/2808.

[49] "Vmax bug in angr," https://github.com/angr/angr/issues/2809.

[50] "Vmul bug in angr," https://github.com/angr/angr/issues/2810.

[51] "Vcvt bug in angr," https://github.com/angr/angr/issues/2829.

[52] "Examiner," https://github.com/valour01/examiner.

[53] "ARM Exploration tools," https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools.

[54] A. Patel, F. Afram, and K. Ghose, "Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors," in *1st International Qemu Users' Forum*. Citeseer, 2011, pp. 29–30.

[55] S.-T. Shen, S.-Y. Lee, and C.-H. Chen, "Full system simulation with qemu: An approach to multi-view 3d gpu design," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. IEEE, 2010, pp. 3877–3880.

[56] K. Tam, A. Fattori, S. Khan, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *NDSS Symposium 2015*, 2015, pp. 1–15.

[57] H. Shi, A. Alwabel, and J. Mirkovic, "Cardinal pill testing of system virtual machines," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 271–285.

[58] "Capstone," https://www.capstone-engine.org/.

[59] "Z3Prover," https://github.com/Z3Prover/z3.

[60] "ARM SIMD Instructions," https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/What-is-SIMD-/ARM-SIMD-instructions.

[61] "ARM WFE Instruction," https://developer.arm.com/documentation/ddi0360/e/programmer-s-model/additional-instructions/wait-for-event-wfe.

[62] Xu Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "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*, 2008.

[63] "Panda.re," https://panda.re/.

[64] "Suterusu," https://github.com/mncoppola/suterusu.

[65] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim, "Fuzzification: Anti-fuzzing techniques," in *Proceedings of the 28th USENIX Security Symposium*, 2019.

[66] E. Güler, C. Aschermann, A. Abbasi, and T. Holz, "Antifuzz: Impeding fuzzing audits of binary executables," in *Proceedings of the 28th USENIX Security Symposium*, 2019.

[67] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte *et al.*, "Isa semantics for armv8-a, risc-v, and cheri-mips," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, 2019.

[68] O. Sahin, A. K. Coskun, and M. Egele, "Proteus: Detecting android emulators from instruction-level profiles," in *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*. Springer, 2018, pp. 3–24.

[69] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-pro: Innovations in test program generation for functional processor verification," *IEEE Design & Test of Computers*, vol. 21, no. 2, pp. 84–93, 2004.

[70] B. Campbell and I. Stark, "Randomised testing of a microprocessor model using smt-solver state generation," *Science of Computer Programming*, vol. 118, pp. 60–76, 2016.

[71] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin, "X-gen: A random test-case generator for systems and socs," in *Seventh IEEE International High-Level Design Validation and Test Workshop, 2002.* IEEE, 2002, pp. 145–150.

[72] T. Bauereiss, B. Campbell, T. Sewell, A. Armstrong, L. Esswood, I. Stark, G. Barnes, R. N. Watson, and P. Sewell, "Verified security for the morello capability-enhanced prototype arm architecture," in *European Symposium on Programming*. Springer International Publishing Cham, 2022, pp. 174–203.

[73] Y. Katz, M. Rimon, and A. Ziv, "Generating instruction streams using abstract csp," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, pp. 15–20.

[74] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1286–1303.

[75] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, 1998.

[76] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM SIGPLAN Notices*, 2014.

[77] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," *ACM SIGPLAN Notices*, 2015.

[78] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016.

[79] J. Jung, H. Hu, J. Arulraj, T. Kim, and W. Kang, "Apollo: Automatic detection and diagnosis of performance regressions in database systems," *Proceedings of the VLDB Endowment*, 2019.

[80] D. Winterer, C. Zhang, and Z. Su, "Validating smt solvers via semantic fusion," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[81] ——, "On the unusual effectiveness of type-aware operator mutations for testing smt solvers," *Proceedings of the ACM on Programming Languages*, no. OOPSLA, 2020.

[82] T. Kapus and C. Cadar, "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*, 2017.

[83] T. Kuchta, T. Lutellier, E. Wong, L. Tan, and C. Cadar, "On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in pdf readers and files," *Empirical Software Engineering*, 2018.