PROTEUS: Detecting Android Emulators from Instruction-level Profiles

Onur Sahin(✉), Ayse K. Coskun, and Manuel Egele

Boston University, Boston, MA, 02215, USA
sahin@bu.edu

Abstract. The popularity of Android and the personal information stored on these devices attract the attention of regular cyber-criminals as well as nation state adversaries who develop malware that targets this platform. To identify malicious Android apps at a scale (e.g., Google Play contains 3.7M Apps), state-of-the-art mobile malware analysis systems inspect the execution of apps in emulation-based sandboxes. An emerging class of evasive Android malware, however, can evade detection by such analysis systems through ceasing malicious activities if an emulation sandbox is detected. Thus, systematically uncovering potential methods to detect emulated environments is crucial to stay ahead of adversaries. This work uncovers the detection methods based on discrepancies in instruction-level behavior between software-based emulators and real ARM CPUs that power the vast majority of Android devices. To systematically discover such discrepancies at scale, we propose the PROTEUS system. PROTEUS performs large-scale collection of application execution traces (i.e., registers and memory) as they run on an emulator and on *accurate software models of ARM CPUs*. PROTEUS automatically identifies the instructions that cause divergent behavior between emulated and real CPUs and, on a set of 500K test programs, identified 28K divergent instances. By inspecting these instances, we reveal 3 major classes of root causes that are responsible for these discrepancies. We show that some of these root causes can be easily fixed without introducing observable performance degradation in the emulator. Thus, we have submitted patches to improve resilience of Android emulators against evasive malware.

1 Introduction

Android is a fast growing ecosystem. By acting as a trusted medium between developers and users, application repositories (e.g., Google Play Store) have enabled explosive growth in the number of mobile applications available to billions of users worldwide [6]. Currently, the Play Store consists of more than 3.7M Android applications with thousands of new applications emerging every day [9]. Unfortunately, this massive ecosystem is also appealing to miscreants who seek to infect a wide set of users with malicious applications.

To protect users, malware analysis systems are widely used in both academia and industry. Since malware can easily defeat static analysis via obfuscation and packing [14], contemporary analysis systems for Android adopt dynamic analysis to inspect the runtime behavior of applications. State-of-the-art malware analyzers for Android are based on emulators [23, 28, 30], which can easily scale across multiple hosts to inspect vast number of Android apps. Such emulation-based analysis also offers easy instrumentation [30] and fast state restore capabilities (e.g., orders of magnitude faster than bare-metal [22]), making the emulation-based analysis approach appealing to security researchers and practitioners.

The effectiveness of these dynamic malware analysis systems, however, is largely at risk due to an emerging class of evasive malware. Such malware looks for discrepancies that exist between emulated and real systems before triggering any malicious attempt. By ceasing malicious activities on an emulated enviroment, the malware can thwart existing emulator-based malware analyzers. The situation is alarming as studies show a rising number of malware instances that employ evasion tactics [18] (e.g., Branco et al. find evasion methods in more than 80% of 4M malware samples [13]). For Android, several recent classes of evasive malware (e.g., *Xavier* [1], *Grabos* [7]) have already been identified in the Play Store. A crucial step for defending against such malware is to systematically extract the discrepancies between emulated and real systems. Once discovered, such discrepancies can be eliminated [19] or can be used to inspect applications for presence of evasion tactics leveraging these artifacts [13].

Many of the approaches to date [10, 25, 29] discover discrepancies of emulation-based sandboxes in an ad hoc fashion by engineering malware samples or specific emulator components (e.g., scheduling). Such manual approaches cannot provide large-scale discovery of unknown discrepancies, which is needed to stay ahead of adversaries. Recent work [17] automatically identifies file system and API discrepancies used by several Android malware (e.g., [1, 7]). Evasion tactics that rely on such artifacts can be rendered ineffective by using modified system images and ensuring the API return values match those in real devices [12]. Besides API/file checks, a malware can also leverage differences in the semantics of CPU instructions to fingerprint emulation [13] (e.g., by embedding checks in the native code [25]). As opposed to ad hoc approaches or API/file heuristics, our work focuses on systematically discovering instruction-level discrepancies that are intrinsically harder to enumerate and fix for modern complex CPUs.

Prior discoveries of instruction-level discrepancies in emulated CPUs are limited to x86 instruction set [21, 24, 27], while the vast majority mobile devices use ARM CPUs. Despite the large number of discrepancies reported in prior work [21, 24], such findings are not readily useful for improving the fidelity of emulators as their analysis does not reveal the root causes of discrepancies. Such analysis of root causes is essential as not all discrepancies are reliable detection heuristics due to `Unpredictable` ARM instructions [4], whose behavior varies across platforms. In addition, reliance on physical CPUs to obtain the ground truth instruction behavior poses practical limitations on the number of test cases (e.g., instructions, register/memory operands, system register settings) that can be covered. Approaches to improve coverage [27] are based on heavy analysis of ISA specifications, which are notorious for their complexity and size.

To address the shortcomings above and identify instruction-level discrepancies in Android emulators at a scale, we propose to collect and analyze a large number of instruction-level traces corresponding to execution on real ARM CPUs and emulators. By recording how each ARM instruction modifies the architectural state (i.e., registers and memory) on an emulated and real ARM CPU, we can automatically detect divergences that are directly observable by user-level programs. To scale the divergence analysis system, we demonstrate the feasibility of using accurate software models for ARM CPUs instead of physical hardware.

We build our instruction-level analysis framework into a new system, Proteus. Proteus automatically identifies architectural differences between real and emulated ARM CPUs. Proteus uses official software models for ARM CPUs (i.e., *Fast Models* [3]) to gather detailed and accurate instruction-level traces corresponding to real CPU operation. We instrument QEMU to collect traces for emulated CPUs. We target QEMU as it forms the base of state-of-the-art Android malware analysis systems [23, 28, 30] as well as the Android SDK emulator. We evaluate our system with over a million CPU instructions. Our randomized test cases allow us to examine instruction behavior that would not be triggered during execution of conventional compiler-generated programs.

Proteus automatically groups the instructions that generate similar divergent behavior and reveals several major classes of instruction-level discrepancies between emulated and real ARM CPUs. We find that a single root cause (e.g., relaxed opcode verification) can account for a large number divergent cases and that some of these sources of divergences can be eliminated by minor modifications in the QEMU source code. To improve resilience of Android emulators against *detection via CPU semantic attacks*, we have disclosed our root cause findings including patches where appropriate to the QEMU community[1]. Our evaluation of discovered discrepancies on physical devices and SDK emulators demonstrates how unprivileged user-mode programs can deterministically fingerprint Android emulators to easily perform CPU semantic attacks (e.g., by using a few CPU instructions in native code). To the best of our knowledge, this is the first systematic study to demonstrate instruction semantic attacks against QEMU's ARM support. Overall, we make the following specific contributions:

- **Proteus:** We design, implement, and evaluate a scalable approach for discovering discrepancies between emulated and real ARM CPUs (Section 3). Our system collects a large number of instruction-level traces from accurate software models of ARM CPUs and from an instrumented QEMU instance. Proteus automatically identifies the instructions and conditions that cause a divergence, and groups instructions with similar behavior to facilitate further inspection for root cause analysis (Section 4).
- **Novel Attack Surface:** We systematically analyze the divergences found by Proteus and uncover novel detection methods for Android emulators based on instruction-level differences between emulated and real ARM CPUs (Section 5.1). We show the effectiveness of these methods for deterministically distinguishing physical devices from Android emulators (Section 5.3).
- **Fidelity Improvements:** We identify a set of root causes (Section 5.2) that are responsible for a large set of divergences. We show that some of these root causes can be eliminated in Android emulators through minor fixes without causing any observable performance overhead (Section 5.4).

[1] We have eliminated several root causes as part of our work and have already submitted a patch.

2 Background

This section provides a brief overview of the ARM architecture and clarifies the terminology that we use throughout the rest of this paper. We also describe the attack model we are assuming in this work.

2.1 ARMv7-A Architecture

This paper focuses on ARMv7-A instruction set architecture (ISA), the vastly popular variant of ARMv7 that targets high-performance CPUs which support OS platforms such as Linux and Android (e.g., smartphones, IoT devices). The ARM architecture implements a Reduced Instruction Set Computer (RISC) organization where memory accesses are handled explicitly via load/store instructions. Each ARM instruction is of fixed 32-bit length. ARMv7-A features 16 32-bit registers (i.e., 13 general purpose registers (`R0-R12`), stack pointer (`SP`), link register (`LR`), program counter (`PC`)) accessible in user-mode (`usr`) programs. The CPU supports 6 operating modes (`usr`,`hyp`,`abt`,`svc`,`fiq`,`irq`) and 3 privilege levels `PL0`, `PL1` and `PL2` (i.e., lower numbers correspond to lower privilege levels). The *Current Program Status Register* (`CPSR`) stores the CPU mode, execution state bits (e.g., endianness, ARM/Thumb instruction set) and status flags.

Undefined Instructions: The ARMv7 specification explicitly defines the set of encodings that do not correspond to a valid instruction as architecturally `Undefined`. For example, Figure 1 shows the encoding diagram for multiplication instructions in ARMv7. The architecture specification [4] states that the instructions are `Undefined` when the `op` field equals 5 or 7 in this encoding.

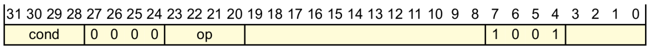

Fig. 1: Encoding diagram for multiplication instructions in ARMv7 ISA [4].

An `Undefined` instruction causes the CPU to switch to the undefined (`und`) mode and generates an undefined instruction exception. An undefined instruction exception is also generated when an instruction tries to access a co-processor that is not implemented or for which access is restricted to higher privilege levels [4].

Unpredictable Instruction Behavior: The ARM architecture contains a large set of instruction encodings for which the resulting instruction behavior is unspecified and cannot be relied upon (i.e., `Unpredictable`). ARM instructions can exhibit `Unpredictable` behavior depending on specific cases of operand registers, current CPU mode or system control register values [4]. For example, many instructions in the ARM architecture are `Unpredictable` if the `PC` is used as a register operand. In addition, some instruction encoding bits are specified as *"should be"* and denoted as "(0)" and "(1)" in ARM's official encoding diagrams. While different encodings for *"should be"* bits do not correspond to different instructions, the resulting behavior is `Unpredictable` if a given encoding fails to match against the specified *"should be"* bit pattern.

The effect of an `Unpredictable` instruction is at the sole discretion of the CPU manufacturer and can behave as a `NOP` or `Undefined` instruction, or can change the architectural state of CPU. Consider the "`LDMDA pc!,{r0,r1,r5,r6, r8,sp,lr}`" `Unpredictable` instruction (encoded as `0xE83F6163`), which loads

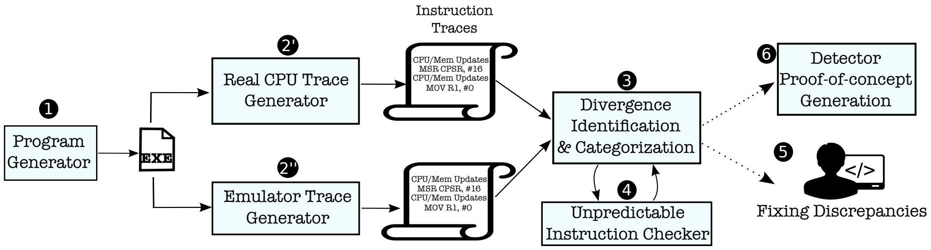

Fig. 2: Overview of Proteus.

the given set of registers from consecutive memory addresses starting at PC and writes the final target address back to PC. This instruction causes undefined instruction exception on a real CPU while it modifies the PC and causes an infinite loop on QEMU. Note that both behaviors comply with the ARM specification.

2.2 Threat Model

The aim of the malware author is to evade detection by the analysis tools and distribute a malicious application to real users. The malware possesses a set of detection heuristics to distinguish emulators from real devices. Malware achieves evasion by ceasing any malicious behavior on an emulated analysis environment, which could otherwise be flagged by the analysis tool. Once the malware escapes detection and reaches real users, it can execute the harmful content within the application or dynamically load the malicious payload at runtime [26].

Our work focuses on discrepancies that are observable by user-level programs. Thus, we assume applications running in `usr` mode at the lowest `PL0` privilege level. Since our technique detects emulators by natively executing CPU instructions and monitoring their effects, we assume an Android application that contains a native code. This is a common case for many applications (e.g., games, physics simulations) that use native code for the performance-critical sections and for the convenience of reusing existing C/C++ libraries [2, 26].

We assume that applications are subject to dynamic analysis in a QEMU-based emulation environment. Indeed, state-of-the-art dynamic analysis frameworks that are commonly used in academia [28, 30] and industry [23] use QEMU as the emulation engine. In addition, the Android emulator that is distributed with the Android SDK is also based on QEMU.

3 Proteus System Architecture

The aim of the proposed Proteus system (Figure 2) is to find the differences in semantics of instructions executed on a real and an emulated ARM CPU. Proteus consists of a trace collection part and an analysis component to automatically identify and classify divergences. This section provides an overview of the core components of Proteus and describes its high-level operation.

Central to our system is collection of detailed instruction-level traces that capture the execution behavior of programs on both emulated and real CPUs. The traces capture all updates to user-visible registers as well as the operands in memory transactions from load/store instructions. If a program terminates by a CPU exception, the respective signal number is also recorded.

The *"Program Generator"* component (❶) generates the test programs which are used for collecting instruction-level traces and discovering discrepancies. Note that ARM CPU emulation in QEMU is inadvertently tested using millions of apps by Android developers. Thus, programs generated for divergence identification should also exercise platforms for uncommon cases beyond the set of instructions emitted by compilers and found in legitimate Android apps.

For each generated test program, we collect its instruction-level traces by executing the same binary on two different platforms (❷) which provide the traces corresponding to execution on an emulator and a real CPU.

The *"Divergence Identification & Categorization"* component (❸) compares emulator and real CPU traces of a program to identify the initial point of divergence. A divergence can be due to a mismatch in register values, memory operands or exception behavior. Divergent cases that stem from the same mismatch are grouped together automatically to facilitate manual inspection of discovered discrepancies. Our hypothesis behind the grouping is that there exist a small number of root causes that cause the same divergent behavior (e.g., exception mismatch) on potentially a large set of test cases. For instance, we can group together the divergent instructions that generate an illegal instruction exception in a real CPU but execute as a valid instruction in emulator. We also check if the divergent instruction is `Unpredictable` (❹). Since `Unpredictable` instructions can exhibit different behavior across any two platforms, we do not treat divergences that stem from these instructions as a reliable detection method.

Overall, Proteus provides us with the instruction encoding that caused the divergent behavior, register values before that instruction, divergence group as well as the difference between the traces of emulated and real CPU (e.g., signal number, CPU mode, etc.) which occurs after executing the divergent instruction. We can optionally identify why QEMU fails to faithfully provide the correct behavior as implemented by the real CPU and fix the source of mismatch (❺). Proteus can also generate a proof-of-concept emulation detector (❻), which reconstructs the divergent behavior by setting respective register values, executing the divergent instruction and checking for the resulting mismatch that Proteus identifies during the *"Divergence Identification & Categorization"* stage.

4 Proteus Implementation

In this section, we describe our implementation of the proposed Proteus system for detecting instruction-level differences between emulated and real ARM CPUs. In Section 4.1, we describe our framework for acquiring instruction-level traces. Section 4.2 describes how we use this framework to collect a large number of sample traces and automatically identify discrepancies.

4.1 Instruction-level Tracing on ARM-based Platforms

Collected Trace Information: For our purposes, a trace consists of all general-purpose registers that are visible to user-level programs, which provide a snapshot of the architectural state. Specifically, we record the `R0-R12`, `SP`, `PC`, `LR` and `CPSR` registers (see Section 2). Finally, we record operands of all memory operations. Various ARM instructions can load/store multiple registers sequentially from a base address. We record all the data within the memory transaction as well as the base address. This trace information gives us a detailed program-visible behavior of CPU instructions. Thus, any discrepancy within the trace is visible to a malware and can be potentially leveraged for evasion purposes.

Emulator Traces through QEMU Instrumentation: QEMU dynamically translates the guest instructions (e.g., ARM) for execution on the host machine (e.g., x86). Translation consists of several steps. First, guest instructions within a basic block are disassembled and converted into a platform-agnostic intermediate representation called TCG (*Tiny Code Generator*). Next, generated TCG code blocks (i.e., *translation block*) are compiled into host ISA for execution.

To implement tracing capability in QEMU, we inject extra TCG operations into each translation block during the translation phase. These extra TCG operations dump the trace information during the execution phase. We use the helper functionality within QEMU to generate the extra TCG code. The main use of the helper functionality in QEMU is to allow developers to extend the capabilities of TCG operations for implementing complex instructions. We inject the extra TCG operations for every disassembled instruction to achieve per-instruction tracing granularity. Specifically, we modify the disassembly routines of ARM instructions to inject TCG operations that record registers. We also modify the load/store routines to record address and data values for memory transactions.

We use QEMU 2.7.0 from Android repositories[2], which forms the base of the SDK emulator used in modern Android malware analyzers [23, 28, 30]. QEMU 2.7.0 is the most recent version adopted in current SDK emulators. To ease instrumentation and facilitate the data collection, we use QEMU in user-mode configuration as opposed to full-system emulation. We use full-system SDK emulators during our evaluation of discovered discrepancies (Section 5.3).

Accurate Real CPU Traces using ARM Fast Models: Gathering detailed instruction-level traces from real CPUs is challenging and, due to practical limitations on the number of devices that can be used, does not scale well. In this work, we propose to use accurate functional models of ARM CPUs (i.e., *Fast Models* [3]) to obtain traces corresponding to execution on real CPUs. Fast Models are official software models developed and maintained by ARM and provide complete accuracy of software-visible semantics of instructions.

ARM Fast Models provide a set of trace sources which generate a stream of trace events when running the simulation. Once a target set of trace sources are specified, Fast Models emit trace events whenever a change occurs on a trace source. These trace events are provided over a standardized interface called *Model Trace Interface (MTI)*. We use an existing plugin called *GenericTrace* to record trace events over the MTI interface.

Our work is based on a Cortex-A15 fast model which implements the ARMv7 ISA. We specify "*inst*", "*cpsr*", "*core_ loads*", "*core_ stores*" and "*core_ regs*" trace sources, which capture changes in register values as well as data/address operand values in memory transactions.

[2] https://android.googlesource.com/platform/external/qemu-android/+/qemu-2.7.0

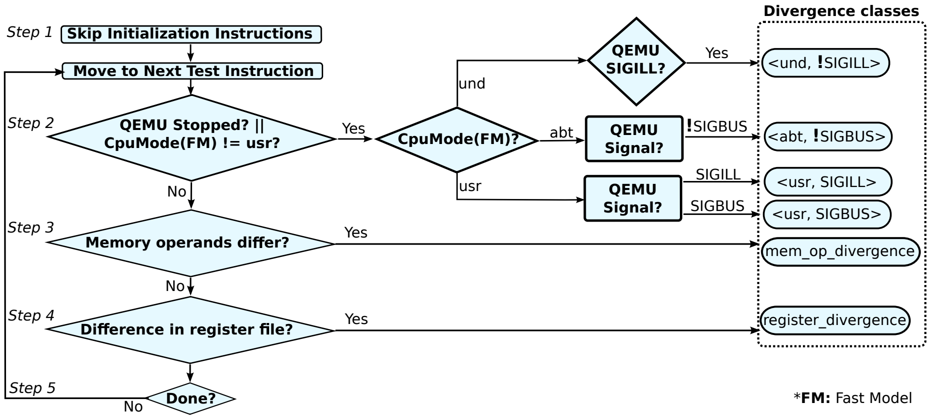

Fig. 3: Illustration of the flow for comparing the Fast Model and QEMU traces.

4.2 Identifying Emulated vs. Real CPU Discrepancies with Tracing

This section describes how we use our tracing capabilities (Section 4.1) to find differences in instruction semantics between emulated and real ARM CPUs.

Generating Test Cases: We generate valid ELF binaries as inputs to our tracing platforms. We choose to use programs that contain random instructions. Specifically, each input binary contains 20 random bytes corresponding to 5 ARM instructions. We use this randomized approach to be able to exercise emulators with uncommon instructions which are not likely to be emitted by compilers. We use more than one instruction per binary to be able to cover more instructions each time a simulation is launched for a test program.

Each test program starts with a few instructions that set the CPU state, clear registers and condition flags. By default, the programs run on the Fast Model in `svc` mode and no stack space is allocated. Thus, we use these initialization instructions to ensure that CPU mode is set to `usr` and `SP` points to the same address on both platforms. We also clear all registers to ensure that programs start from identical architectural state on both emulator and real CPU. These initialization instructions are followed by 5 random instructions. Finally, each test case ends with an `exit` system call sequence (i.e., `mov r7,#1; svc 0x0`).

Identifying Divergence Points: This phase of the Proteus system consumes the traces collected from QEMU and ARM Fast Model to identify and group divergent behaviors. To identify the initial point where QEMU and Fast Model traces of an input program diverge, we perform a step-by-step comparison.

The step-by-step comparison procedure is illustrated in Figure 3. We skip the portion of the traces which corresponds to the initialization instructions described in the previous section (Step 1) to avoid false alarms that arise from the initial state differences between QEMU and Fast Model. We walk through the remaining instruction sequence until either a difference exists in the collected trace data or the test program on QEMU terminates due to an exception. If the program terminates on QEMU or the CPU mode on Fast Models switches to a different mode than `usr`, we examine whether this exception behavior matches between QEMU and real CPU (Step 2). We perform the comparison using the CPU mode from the Fast Model and the signal received by the program upon termination on QEMU. Note that there is no exception handling or signal mechanism on Fast Models as no OS is running. Depending on this CPU mode and signal comparison, we determine whether the observed behavior falls into one of the four possible divergent types below. We use a tuple representation as `<FastModel_response, QEMU_response>` to categorize divergent behavior.

- `<und,!SIGILL>`: This group represents the cases where QEMU fails to recognize an architecturally `Undefined` instruction. If the Fast Models indicate that CPU switches to `und` mode, the expected behavior for QEMU is to deliver a `SIGILL` signal to the target program. This is because execution of an `Undefined` instruction takes the CPU into `und` mode and generates an illegal instruction exception. Thus, the cases where Fast Model switches to `und` mode while QEMU does not deliver a `SIGILL` signal is a sign of divergence.
- `<usr,SIGILL>`: This class of divergence contains cases where QEMU terminates by an illegal instruction signal (`SIGILL`) while Fast Models indicate the target instruction is valid (i.e., cpu remains in `usr` mode).
- `<abt,!SIGBUS>`: This class captures the cases where QEMU fails to recognize a data/prefetch abort and hence does not generate a bus error (i.e., deliver `SIGBUS`). Prefetch aborts are caused by failing to load a target instruction while data aborts indicate that the CPU is unable to read data from memory (e.g., due to privilage restrictions, misaligned addresses etc.) [4].
- `<usr,SIGBUS>`: This divergence type represents the opposite of the previous case. Specifically, QEMU detects a bus error and delivers a `SIGBUS` to the test program while the Fast Models indicate that the memory access made by the target program is valid (i.e., cpu is not in `abt` mode).

If no exception is triggered for an instruction, we further compare the registers and memory operands within the collected trace data. We determine memory operand divergence (Step 3) if the address or the number of transferred bytes differ between QEMU and Fast Model traces. We do not treat data differences as divergence since subtle differences may exist in the initial memory states of QEMU and Fast Models. We drop cases with different memory values from further examination as the loaded data would propagate into register state and cause false positive divergence detection. Finally, if no divergence is identified in exception behavior or in memory operands, we compare the user-level registers (Step 4) to detect any register state divergence. Steps 2-4 presented in Figure 3 continues for the remaining random instructions in the test program.

Since `Unpredictable` instructions can cause different legitimate behaviors on any two CPU implementations, we cannot use these instructions to deterministically differentiate emulators from real systems. Thus, if a divergent instruction identified in Steps 2-4 is `Unpredictable`, we do not classify this case into any divergence group. However, an officially verified tool or a programmatic methodology to check if a given ARM instruction would generate `Unpredictable` behavior is unavailable. Thus, we use an open-source specification of ARMv7 architecture written in Standard ML (SML) and proposed by the earlier work [15].

Generating Detector Programs: Based on the identified divergent instruction semantics, PROTEUS can optionally generate a proof-of-concept program that determines whether the program is executed on an emulator or a real system. This section describes how our system generates these programs and provides insight into how a real-life malware can implement CPU semantic attacks.

PROTEUS generates detector programs by rewriting a set of pre-built template binaries to reconstruct the divergent behavior. The template programs simply execute the divergent instruction and check whether the resulting effect matches with QEMU or real CPU behavior. Before executing the divergent instruction, we set up the environment with the necessary changes to trigger divergent behavior and observe its effect. These changes are (1) setting the register values (`CPSR` and `R0-R12`) with the values provided from the divergence identification phase, (2) installing signal handlers for exception-related divergent behavior. We use `sigsetjmp`/`siglongjmp` provided by the standard C library to preserve the CPU state which would otherwise be altered by the changes we perform. Listing 1.1 illustrates an example detector program for the `<abt,!SIGBUS>` group. The current CPU state is saved with `sigsetjmp` on line 11. The register state is loaded with target values (line 13) and the divergent instruction (`LDM r7,{r4}`) is executed (line 15) to reconstruct the divergent behavior. The program determines emulation if it does not receive a `SIGBUS` after the divergent instruction accesses the misaligned memory address stored in `R4`. The original CPU state before constructing the divergent behavior is restored on either line 7 or 16. We simply build one template program for each of the six divergence groups. Depending on the divergence group, for a given divergent instruction, we pick the corresponding sample template to rewrite a new detector program.

```
 1  sig_atomic_t sig_info = 0;
 2  sigjmp_buf buf;
 3
 4  void catch_signals(int signum) {
 5    if (signum == SIGBUS)
 6      sig_info = 1;
 7    siglongjmp(buf, 1);
 8  }
 9  sig_atomic_t say_my_name() {
10    prepare(); // register signal handlers
11    if(!sigsetjmp(buf,1)) {
12      // Set target registers
13      asm("LDR  r4,=0x00008075");
14      // execute divergent instruction
15      asm(".byte 0x10,0x00,0x97,0xe8");
16      siglongjmp(buf, 1);
17    }
18    cleanup(); // remove signal handlers
19    return sig_info;
20  }
21  void main() {
22    sig_atomic_t ret = say_my_name();
23    (ret==1) ? printf("real cpu"):printf("emulator");
24  }
```

Listing 1.1: A sample program snippet for detecting Android emulator.

5 Evaluation

This section demonstrates the capabilities of PROTEUS for identifying the discrepancies of QEMU's emulated CPU from a real ARM CPU. We systematically analyze the divergences reported by PROTEUS to identify the root causes of the discrepancies. On a real smartphone and Android emulator, we demonstrate how our findings can fingerprint the underlying platform. Finally, we demonstrate the feasibility of fixing several root causes of divergences without any observable performance penalty. Overall, we seek to answer the following questions:

- Are there any observable discrepancies between an emulated and real CPU? If so, how prevalent are these differences? (Section 5.1)
- How effective are the divergences reported by PROTEUS in terms of fingerprinting real hardware and dynamic analysis platforms? (Section 5.3)

– What are the root causes of the discrepancies (Section 5.2) and can we eliminate them in QEMU without impacting its performance? (Section 5.4)

5.1 Divergence Statistics from Proteus

In order to address our first research question, we use Proteus to examine the instruction-level traces from 500K input test programs. Figure 4 shows the number of instructions executed in the test programs until a divergence occurs or QEMU stops due to an exception. The majority of the test cases (45%) finish after a single instruction only and, almost all test cases (>94%), either diverge or cause an exception on QEMU after executing the 5 instructions in our test programs. Overall, our system analyzed over 1.06M CPU instructions. Table 1 presents an overall view of the results by Proteus showing a comparison between QEMU and Fast Models in terms of the exception behavior (Table 1a) as well as extent of divergences per group (Table 1b).

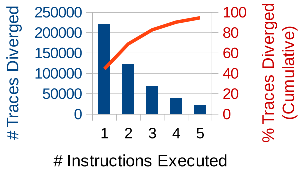

Fig. 4: #Instructions before divergence or exception.

Fast Model Behavior	QEMU Response		
	SIGILL	SIGSEGV	None
und	149436 (30%)	826 (0.2%)	9341 (1.9%)
abt	11 (0.002%)	12471 (2.5%)	528 (0.1%)
svc	0	18932 (3.8%)	14609 (2.9%)
usr	5270 (1.1%)	176975 (35%)	0

(a) Exception behavior comparison.

<und, !SIGILL>	10167 (2%)
<usr, SIGILL>	5270 (1.1%)
<abt, !SIGBUS>	13010 (2.6%)
<usr, SIGBUS>	0
mem_op_difference	200 (0.05%)
register_divergence	12 (0.002%)

(b) Divergences by type.

Table 1: Divergence statistics generated by Proteus for 500K test cases containing 2.5M random ARM instructions. Remaining instances of 500K programs (not shown in the Table 1a) are (1) 83,125 (17%) cases due to `Unpredictable` instructions, (2) 27,048 (5.4%) non-divergent cases where programs finish successfully on both platforms and (3) 1216 cases that differ due to memory values. Note that we do not treat these 3 cases as divergent (see Section 4.2).

Table 1a presents a summary of the cases where either QEMU terminates the program or the CPU mode changes in Fast Models. Overall, we observe two types of signals in QEMU (i.e., `SIGILL`, `SIGSEGV`) and CPU mode in Fast Models cover `und`, `abt`, `svc` and `usr` modes. `None` represents the cases where QEMU does not generate an exception. Most instances correspond to illegal instruction (`<und, SIGILL>`) and valid memory access (`<usr, SIGSEGV>`) cases in which the behavior in QEMU complies with Fast Models (i.e., not divergent). A large number of instances are *Supervisor Call* (`svc`) instructions which cover a large encoding space in ARM ISA. `svc` instructions are used to request OS services and are not a major point of interest for our work as we focus on the discrepancies that are observable in the user space. In Table 1a, such non-divergent cases are highlighted in gray. The remaining instances in Table 1a, along with the non-exception related differences (i.e., memory operand and register) are grouped into the divergence types as per the methodology described in Section 4.2.

Table 1b provides the number of instances per each divergence type. The largest number of divergences (i.e., 2.6% of 500K test programs) belong to `<abt, !SIGBUS>` group which hints that QEMU does not correctly sanitize the invalid memory references that cause data/prefetch aborts in CPU. Proteus also finds a large number of instructions that are recognized as architecturally `Undefined` only by the Fast Models (i.e., `<und, !SIGILL>` group). These point to cases where QEMU does not properly validate the encoding before treating the instruction as architecturally valid. We also find a large number of instructions which are detected as illegal only by QEMU, executing without raising an illegal instruction exception on the Fast Model (i.e., `<usr, SIGILL>` group). Proteus also finds a smaller number of cases (i.e., 0.05%) with divergent register update or memory operation which correspond to `register_divergence` and `mem_op_difference` groups in Table 1b, respectively. These examples hint at cases where the implementation of a valid instruction contains potential errors in QEMU, causing a different register or memory state than on a real CPU. Overall, despite the significant testing of QEMU, we observe that there are still many divergences where QEMU does not implement the ARM ISA faithfully.

5.2 Root Cause Analysis

While the Proteus system can identify large numbers of discrepancies between real and emulated ARM CPU, it does not pinpoint the root causes in QEMU that lead to a different behavior than ground truth (i.e., Fast Model behavior). This section presents our findings from an analysis of root causes of divergent behavior in QEMU. This analysis gives us, compared to large number of divergences identified, a smaller set of unique errors in QEMU that lead to divergence on a wide set of programs (Table 1b). Analyzing the root causes also allows us to pinpoint implementation flaws and devise fixes (Section 5.4).

In our analysis, for a divergence group, we first identify common occurrences in the bit fields *[27:20]* of a divergent 32-bit instruction encoding. In the ARM architecture, these bits contain opcodes that are checked while decoding the instruction on QEMU and real CPU. We identify the instructions with the most commonly occuring opcodes to (1) consult the ISA specification to check how these instruction should be decoded and (2) check how QEMU processes these instruction. We determine the root cause of the discrepancy by manually analyzing QEMU's control flow while executing a sample of these instructions. Once we examine the source of discrepancy (e.g., a missing check, an unimplemented feature of QEMU), we remove all possible encodings that stem from the same root cause from our statistics to find other unique instances of errors in QEMU.

Through this iterative procedure, we identified several important classes of flaws in QEMU that result in a different instruction-level behavior than a real CPU. We discuss some of our findings in the following paragraphs.

Incomplete Sanitization for Undefined Instructions: We discover that QEMU does not correctly generate illegal instruction exception for a set of `Undefined` instruction encodings. These cases are identified from the `<und, !SIGILL>` group provided by Proteus. Thus, a malware can achieve evasion simply by executing one of these instructions and ceasing malicious activity if no illegal instruction exception is generated.

We find that this particular group of divergences arises as QEMU relaxes the number of checks performed on the encoding while decoding the instructions. For instance, the ARM ISA defines a set of opcodes for which the synchronization instructions (e.g., `SWP`, `LDREX`) are `Undefined`, and thus should generate an illegal instruction exception. However, QEMU does not check against these invalid opcodes while decoding the synchronization instructions, causing a set of `Undefined` encodings to be treated as a valid `SWP` instruction. In fact, we identified 715 divergent test cases which are caused by this missing invalid opcode check for the `SWP` instruction. In Table 2, we provide the encoding and the conditions that cause divergent behavior for this `SWP` instruction example as well as other similar errors in QEMU that we have identified.

<table>
<tr><th>Instruction Encoding (cond $\in [0, 0xE]$)</th><th>Divergent Condition</th><th>QEMU Behavior</th><th>Real CPU Behavior</th><th>#Cases</th></tr>
<tr><td>cond:4|0001|op:4|*:12|1001|*:4</td><td>op = 1,2,3,5,6,7</td><td>SWP Inst.</td><td>Undefined</td><td>715</td></tr>
<tr><td>cond:4|1100010|*:9|101|*:1|op:4|*:4</td><td>op != 1,3</td><td>64-bit VMOV</td><td>Undefined</td><td>424</td></tr>
<tr><td>cond:4|11|op:6|*:20</td><td>op = 1,2</td><td>VFP Store</td><td>Undefined</td><td>51</td></tr>
<tr><td>cond:4|11101|*:2|0|*:8|1011|*:1|op:2|1|*:4</td><td>op = 2,3</td><td>VDUP Inst.</td><td>Undefined</td><td>3</td></tr>
<tr><td>cond:4|110|op:5|*:8|101|*:9</td><td>op != 4,5,8-25, 28,29</td><td>VFP Store</td><td>Undefined</td><td>2</td></tr>
</table>

Table 2: Several `Undefined` instruction encodings that are treated as valid instructions by QEMU. ":X" notation represents the bit length of a field while "*" represents that the field can be filled with any value (i.e., 0 or 1).

During our root cause analysis, we find that a large portion of the instances in `<und, !SIGILL>` group (87%) are due to instructions accessing the co-processors with ids 1 and 2. These co-processors correspond to FPA11 floating-point processor that existed in earlier variants of the ARM architecture while newer architectures (>ARMv5) use co-processor 10 for floating point (VFP) and 11 for vector processing (SIMD). While accesses to co-processors 1 and 2 are `Undefined` on a real CPU, QEMU still supports emulation of these co-processors [8]. Thus, these instructions generate an illegal instruction exception only on the real CPU.

Misaligned Memory Access Checks: As hinted by Proteus with the large number of instances in the `<abt, !SIGBUS>` group in Table 1b, we identify that QEMU does not enforce memory alignment requirements (e.g., alignment at word boundaries) for the ARM instructions that do not support misaligned memory accesses. The data aborts caused by such misaligned accesses would take the CPU into `abt` mode and the program is expected to be signalled with `SIGBUS` to notify that the memory subsystem cannot handle the request. Due to missing alignment checks in QEMU, a malware can easily fingerprint emulation by generating a memory reference with a misaligned address and observing whether the operation succeeds (i.e., in QEMU) or fails (i.e., on a real system).

The ARMv7 implementations can support misaligned accesses for the load-/store instructions that access a single word (e.g., `LDR`, `STR`), a half-word (e.g., `LDRH`, `STRH`) or only a byte of data (e.g., `LDRB`, `STRB`). However, other instructions that perform multiple loads/stores (e.g., `LDM`, `STM`) or memory-register swaps for synchronization (e.g., SWP, LDREX, STREX) require proper alignment of the data being referenced. The alignment requirement can be word, half-word or double-word depending on the size of data being accessed by the instruction.

We demonstrate in Section 5.3 how the divergence due to missing alignment enforcements in QEMU can enable evasion in a real-world scenario.

Updates to Execution State Bits: By analyzing the divergent instructions reported by Proteus within the register_divergence group, we identified another important root cause in QEMU due to masking out of the execution state bits during a status register update. Specifically, we analyzed the cases where execution state bits within CPSR differ after an MSR (move to special registers) instruction. Execution state bits in CPSR determine the current instruction set (e.g., ARM, Thumb, Jazelle) and the endianness for loads and stores. While MSR system instructions allow to update CPSR, writes to execution state bits are not allowed with the only exception being the endianness bit (CPSR.E). The ARM ISA specifies that "CPSR.E bit is writable from any mode using an MSR instruction" [4]. However, since updates on the CPSR.E bit by an MSR instruction are ignored in current QEMU, software can easily fingerprint the emulation by simply trying to flip this bit (e.g., using MSR CPSR_x, 0x200 instruction) and checking whether the endianness has been succesfully changed.

Observations from other statistics: Our initial investigations on <usr, SIGILL> and mem_op_divergence groups did not reveal any further root causes as above. We find that the majority of the divergent cases in mem_op_divergence group (>97%) are due to VFP/SIMD instructions effecting the extension registers. Our current work focuses on the user-mode general purpose registers only. During analysis on <usr, SIGILL> group, we identified divergences due to Unpredictable instructions. This issue is due to the incomplete SML model [15] which misses some Unpredictable instructions in our test cases (Figure 2). For instance, we find that 761 divergence cases in <usr, SIGILL> group are due to Unpredictable encodings of a PLD (i.e., preload data) instruction, which behave as a NOP in Fast Model but generate an illegal instruction exception in QEMU.

5.3 Demonstration with Real Smartphones and the SDK Emulator

In this section, we address our second research question on evaluating the effectiveness of the divergences found by Proteus for real-world emulation detection. To tackle this objective, we evaluate the divergences described in Section 5.2 on a physical mobile platform and Android emulator. We use Nexus 5 (ARMv7) and Nexus 6P (ARMv8) smartphones as our real hardware test-beds and use the full-system emulator from the Android SDK. We choose the SDK emulator as it has been a popular base for Android dynamic analysis frameworks [23, 28, 30].

Evaluating Unsanitized Undefined Encodings: We use the detection binaries generated by Proteus to evaluate the Undefined instructions that are incompletely sanitized in QEMU (i.e., <und, !SIGILL> group). These cases are expected to generate an illegal instruction exception only on a real CPU.

We find that the SDK's copy of QEMU does not incorporate the FPA11 floating point co-processor emulation which is supported in our version of QEMU and accessed by the instructions that use co-processors 1 and 2. Thus, these instructions are `Undefined` in SDK emulator as well and we cannot successfully distinguish the emulator from the real hardware. As discussed in Section 5.1, FPA11 instructions account for 87% of the cases in `<und, !SIGILL>` group. However, we can successfully fingerprint the SDK emulator using all the other divergent `Undefined` instructions. Specifically, all the encodings described in Table 2 can deterministically distinguish between SDK emulator and Nexus 5. The detector programs (Section 4.2) simply register a set of signal handlers and detect the SDK emulator if the program does not receive `SIGILL` upon executing the divergent `Undefined` instruction.

```
/* Put some known data into memory */
int *ptr = calloc(1,sizeof(int));
ptr[0] = 0x12345678;
asm("mov r8,%0" : : "r"(ptr));

/* Read ptr[0] with CPSR.E set to 1 */
asm("msr CPSR_x, #0x200\n\t");
asm("ldr r4,[r8]\n\t");
asm("msr CPSR_x, #0x000\n\t");

asm("mov %0, r4" : "=r"(val) : : );
printf("0x%08X\n", val);
```

Listing 1.2: PoC for emulator detection by flipping endianness bit.

```
/* Put some known data into memory */
int *ptr = calloc(1,sizeof(int));
ptr[0] = 0x12345678;

//Shift address to a non-word boundary
ptr = (int*)((char*)ptr + 0x1);

//Try to read from misaligned address
asm("mov r3,%0" : : "r"(ptr));
asm("LDM r3,{%0}": "=r"(val) : : );

printf("0x%08X\n", val);
```

Listing 1.3: PoC for emulator detection by misaligned memory read.

Evaluating Missing Alignment Checks and Endianness Support: We also show that we can successfully detect the SDK emulator by leveraging the ignored endianness bit updates as well as the missing memory address alignment checks. Listing 1.2 provides a proof-of-concept (PoC) code sample that fingerprints emulation by flipping the endianness bit in the CPU (i.e., `CPSR.E`) and performing a load operation on a known data value to determine whether the endianness has been changed. Executing this code snippet on a real hardware (i.e., Nexus 5 in our case) reads the array value as `0x78563412` instead of `0x12345678` as the `CPSR.E` bit is set to switch from little-endian to big-endian operation for data accesses. However, since the `CPSR.E` bit update is ignored in QEMU, the `LDR` instruction reads the array element into `R4` as `0x12345678` on the SDK emulator. Thus, a malware can easily fingerprint emulation by simply checking the value of target register (i.e., `R4` in this example). Similarly, Listing 1.3 illustrates how the missing alignments checks in QEMU can be leveraged to fingerprint emulation. We shift the word-aligned `ptr` pointer by one byte to create a misaligned reference address. Reading from this misaligned pointer with an `LDM` (i.e., load multiple) instruction causes a bus error (program receives `SIGBUS`) on a real hardware while it succesfully reads the high-order part (`0x123456`) of the target address on the SDK emulator. Thus, a malware can simply determine emulation depending on whether a `SIGBUS` signal is received (i.e., on a real system) upon intentionally causing a misaligned memory access.

Evaluation on a ARMv8 CPU: The 64-bit ARMv8 architecture, which is used in recent smartphones, is compatible with ARMv7. Thus, the CPU semantic attacks we demonstrate in this work also apply to devices powered with ARMv8

CPUs (e.g., Nexus 6P). We evaluated PoC detectors for each root cause we discovered (i.e., Table 2, Listings 1.2 and 1.3) on a Nexus 6P smartphone and successfully distinguished this device from the SDK emulator as well.

5.4 Improving the Fidelity of QEMU

With the capabilities of Proteus for identifying and classifying divergences in instruction-level behavior, in this section, we show the feasibility of eliminating the sources of discrepancies to improve QEMU's fidelity.

We have modified the QEMU source code of the SDK emulator to eliminate the top 3 detection methods in Table 2 based on incomplete sanitization of opcodes for `Undefined` encodings. Specifically, based on the ARM ISA specification [4], we fixed the decoding logic of QEMU to verify all opcode fields for these 3 cases and trigger an illegal instruction exception for the `Undefined` encodings. These fixes eliminated 1190 divergent cases in Table 2. Using various CPU benchmarks from MiBench suite [16], in Figure 5, we verified that the minimal extra code needed to perform additional opcode checks does not introduce any measurable performance overhead. We acknowledge, however, that addressing the alignment check and endianness support in QEMU will require more comprehensive changes than the missing opcode checks for `Undefined` encodings.

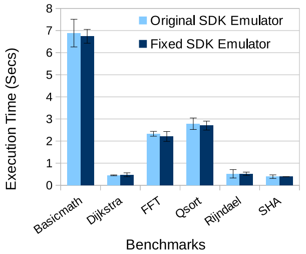

Fig. 5: Overhead evaluation of fidelity enhancements.

6 Discussion and Limitations

Countermeasures: One possible defense against the CPU semantic attacks demonstrated in this work is to, as evaluated in Section 5.4, fix the root causes of instruction-level discrepancies in QEMU. We believe enhancing the fidelity of QEMU is crucial considering the critical role of emulators for Android malware analysis and the growing number of malicious apps that seek to leverage evasion tactics. As a first step towards this objective, we are disclosing our root cause findings and, in fact, have already shared a patch with the QEMU's maintainers.

As Proteus enumerates a set of divergent instructions, similar to prior work that inspects x86 binaries to detect evasion [13], we can scan Android apps for the presence of divergent instructions. Such analysis can be adopted by malware analyzers (e.g., Google's Bouncer [23]) to discover evasive malware that leverages these detection heuristics and prevent them from infecting the Android users.

Another potential countermeasure against the evasive malware that leverages low-level CPU discrepancies is to use real hardware for dynamic analysis instead of emulators [22]. Such a fundamentally different approach can eliminate CPU-level discrepancies. However, practical limitations such as cost, scalability and maintenance inhibits wide-spread adoption of such approaches. In addition, the instrumentation required for analyzing applications on physical devices introduces artifacts itself which allows for fingerprinting [27]. Thus, malware analysis systems for Android will continue to rely on emulators [23, 28, 30].

Limitations: PROTEUS uncovers several classes of observable artifacts in ARM CPU implementations between emulator and real devices. However, there could be other instruction-level discrepancies in current Android emulators that our system could not identify as our scope in this work was limited in several directions. This section discusses these limitations and describes the open-problems.

We demonstrated the capabilities of PROTEUS on the ARMv7 architecture and for the instructions in ARM mode. Recent Android devices also use the latest 64-bit ARMv8 variant of the ISA. Since ARMv8 provides compatibility with ARMv7, as evaluated in Section 5.3, the discrepancies we have discovered in this work also apply to ARMv8 CPUs. Discovering ARMv8-specific discrepancies using PROTEUS simply requires acquiring a Fast Model for an ARMv8 CPU (e.g., Cortex-A53) and repeating the experiments. Our present work did not explore instructions executing in Thumb mode which provides improved code density via 16-bit instructions with limited functionality. Finally, this work focuses on the ARM registers and did not explore potential discrepancies in the extension registers used by VFP/SIMD instructions. Expanding our system to include Thumb instructions and extension registers is part of our immediate future work.

Our present study also does not fully address data-dependent divergences (e.g., depending on the input values from registers or memory). Such limitation is common to fuzzing approaches as exhaustively exploring all possible inputs is computationally infeasible. One approach to improve PROTEUS in this regard would be to repeat the same test cases with several randomized inputs as well as corner cases (e.g., min/max values) as in prior work [21, 27].

As discussed in Sections 5.2 and 5.3, some of the divergences found by PROTEUS are due to `Unpredictable` instructions and do not correspond to an implementation flaw. This is particularly the case as the ARMv7 specification written in SML [15], which we used to check `Unpredictable` instructions, does not cover all `Unpredictable` instruction encodings. A significant contribution of our analysis is that we discovered deterministic CPU-level discrepancies even in the presence of some `Unpredictable` instructions in our test cases. Recently, ARM has released an official machine readable ISA specification written in a domain-specific language named ASL [5]. Unfortunately, the lack of official documentation and tools to work with ASL prevents us from relying on this resource. However, we find ASL specifications a promising future solution for enumerating `Unpredictable` encodings and improving our overall testing methodology.

7 Related Work

This sections overviews prior work on discovering emulation detection methods and explains how PROTEUS distinguishes from or complements them. We also discuss existing defense approaches against evasive malware.

Finding Discrepancies of Emulation Environments: Jing et al. [17] identify a large number of detection heuristics based on the differences in file system entries and return values of Android API calls. For instance, presence of "`/proc/sys/net/ipv4/tcp_syncookies`" file or a `False` return value from the "`isTetheringSupported()`" API implies emulation. Such discrepancies can be easily concealed by editing Android's system images and API implementations to fake real device view [12, 19]. Petsas et al. detect QEMU-based emulation by observing the side effects of caching and scheduling on QEMU [25]. Other work leverages performance side channel due to low graphics performance on emulators to fingerprint emulation [29]. These techniques, however, have practical limitations as they require many repeated trial and observations which increases the detection risk of malware. Our work *systematically* uncovers observable differences in instruction semantics, which achieve deterministic emulation detection through execution of a single CPU instruction.

Similar to our approach, other works also aim at discovering discrepancies of emulators at instruction granularity. Various techniques [21, 24] execute randomized instructions on emulator and real hardware to identify the discrepancies of x86 emulators. To ensure coverage of a wide set of instructions, other work [27] carefully constructs tests cases with unique instructions based on manual analysis of the x86 ISA manual while our technique is fully automated. In addition, the analysis and findings of these studies are limited to x86 instruction set only while the vast majority of mobile devices are powered by ARM CPUs. In addition, these studies classify divergences based on instructions (e.g., using mnemonic, opcodes) which oversees the fact that even different instructions (e.g., `LDM` and `STM`) can diverge due to the same root cause (e.g., missing alignment check). Our study points to the unique root causes in the implementation of CPU emulators. Thus, as we show in Section 5.4, our findings are readily useful for improving the fidelity of QEMU. Finally, as reliance on physical CPUs practically limits the number of test cases (e.g., instructions, register/memory operands, system register settings), we propose a novel scalable system which uses accurate functional models of ARM CPUs (i.e., Fast Models).

Martingoni et al. [20] used symbolic execution traces from a *high-fidelity emulator* to construct test cases that would achieve high coverage while testing a *low-fidelity emulator*. Unavailability of such high-fidelity emulator for Android, however, limits the applicability of this technique for our use.

Defense Against Evasive Malware: Several work proposes to detect divergent behavior in malware as a defense mechanism. Balzorotti et al. [11] detect divergent behavior due to instruction semantics by replaying applications on emulators with the system call sequences gathered from real devices and comparing the runtime behavior. Lindorfer et al. [18] propose a more generic methodology for detecing evasive malware based on the similarity of execution behaviors collected from a set of virtual machines. These approaches do not systematically expose potential causes of divergences that a future malware can use. Our work addresses the problem of proactively finding these instruction-level discrepancies and opens the possibility of pre-emptively fixing them.

Specifically for Android, other works [12, 19] systematically remove observable differences from API calls, file system and properites of emulator devices and demonstrate resistance against evasion. Such approaches, however, require enumeration of root causes of discrepancies. Our Proteus system aids these approaches by enumerating the divergent cases between emulator and real CPUs.

8 Conclusion

Scalable dynamic analysis of Android malware relies on emulators. Due to presence of observable discrepancies between emulated and real systems, however, a malware can detect emulation-based analysis and alter behavior to evade detection. Restoring the effectiveness of Android malware analysis requires systematic approaches to proactively identify potential detection tactics that can be used by malicious authors. This work presented the first systematic study of differences in instruction-level behavior of emulated and real ARM CPUs that power the vast majority of Android devices. We presented the Proteus system for large-scale exploration of CPU semantic attacks against Android emulators. Proteus automatically analyzed detailed instruction-level traces collected from QEMU and accurate software models of ARM CPUs and revealed several major root causes for instruction-level discrepancies in QEMU. We demonstrated the feasibility of enhancing the fidelity of QEMU by fixing the root causes of divergences without any performance impact. We are disclosing our findings and submitted patches to QEMU as a step towards improving QEMU's resiliency against evasive malware.

Acknowledgements

This work was supported by the Office of Naval Research under grants N00014-15-1-2948 and N00014-17-1-2011. We would also like to thank Arm for providing us access to the Fast Models used in this work. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor.

References

1. Analyzing Xavier: An Information-Stealing Ad Library on Android. https://blog.trendmicro.com/trendlabs-security-intelligence/analyzing-xavier-information-stealing-ad-library-android/. [Online].
2. Android Native Development Kit (NDK). https://developer.android.com/ndk/guides/index.html. [Online].
3. ARM Fast Models. https://developer.arm.com/products/system-design/fast-models. [Online].
4. ARMv7-A/R Architecture Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html. [Online].
5. Exploration Tools, A-Profile Architectures. . https://developer.arm.com/products/architecture/a-profile/exploration-tools. [Online].
6. Google has 2 billion users on Android. https://techcrunch.com/2017/05/17/google-has-2-billion-users-on-android-500m-on-google-photos/. [Online].
7. Grabos Malware. https://securingtomorrow.mcafee.com/consumer/consumer-threat-notices/grabos-malware/. [Online].
8. NetWinder Floating Point Notes. http://netwinder.osuosl.org/users/s/scottb/public_html/notes/FP-Notes-all.html. [Online].
9. Number of Android applications. https://www.appbrain.com/stats/number-of-android-apps. [Online].
10. QEMU emulation detection. https://wiki.koeln.ccc.de/images/d/d5/Openchaos_qemudetect.pdf. [Online].
11. D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient detection of split personalities in malware. In *NDSS*, 2010.

12. L. Bordoni, M. Conti, and R. Spolaor. Mirage: Toward a stealthier and modular malware analysis sandbox for android. In *Computer Security – ESORICS*, 2017.
13. R. R. Branco, G. N. Barbosa, and P. D. Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. In *BlackHat*, 2012.
14. M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6:1–6:42, Mar. 2008.
15. A. Fox. Directions in isa specification. In *Interactive Theorem Proving*, 2012.
16. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization (IISWC)*, 2001.
17. Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: Automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, pages 216–225. ACM, 2014.
18. M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti. Detecting environment-sensitive malware. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, pages 338–357. Springer-Verlag, 2011.
19. L. Liu, Y. Gu, Q. Li, and P. Su. Realdroid: Large-scale evasive malware detection on "real devices". In *26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–8, July 2017.
20. L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 337–348, New York, NY, USA, 2012. ACM.
21. L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing cpu emulators. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.
22. S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna. Baredroid: Large-scale analysis of android apps on real devices. In *Annual Computer Security Applications Conference*, ACSAC, 2015.
23. J. Oberheide and C. Miller. Dissecting the android bouncer. *SummerCon*, 2012.
24. R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies (WOOT)*, 2009.
25. T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *European Workshop on System Security (EuroSec)*, pages 5:1–5:6, 2014.
26. S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Network and Distributed System Security Symposium (NDSS)*, 2014.
27. H. Shi, A. Alwabel, and J. Mirkovic. Cardinal pill testing of system virtual machines. In *23rd USENIX Conference on Security Symposium*, 2014.
28. K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.
29. T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIA CCS)*, pages 447–458. ACM, 2014.
30. L. K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *21st USENIX Security Symposium*, pages 569–584, Bellevue, WA, 2012. USENIX.