Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Rethinking anti-emulation techniques for large-scale software deployment

Daehee Jang[a], Yunjong Jeong[a], Sungman Lee[a], Minjoon Park[a], Kuenhwan Kwak[b], Donguk Kim[b], Brent Byunghoon Kang[a,*]

[a] Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea
[b] Samsung Research, Seoul, Republic of Korea

ARTICLE INFO

Article history:
Received 19 September 2018
Revised 5 February 2019
Accepted 10 February 2019
Available online 18 February 2019

Keywords:
Anti emulation
Software analysis
Large scale deployment
Misaligned vectorization
Commercial-off-the-shelf

ABSTRACT

From the security perspective, emulation is often utilized to analyze unknown malware owing to its capability of tracing fine-grained runtime behavior (i.e., execution path exploration). To this end, attackers equip their malware with powerful anti-emulation techniques that fingerprint the emulated system environment, thereby avoiding dynamic analysis. However, this is not the only use case of anti-emulation. Recently, legitimate software vendors are also putting significant efforts to prevent their products running on top of the emulated execution environment. There are mainly two reasons for this which are: (i) securing the intellectual property from emulation-assisted reverse-engineering, and (ii) disallowing the customers using the application without purchasing the actual hardware. From the previous literature, various anti-emulation techniques were explored. Unfortunately, existing techniques are mostly discussed and developed with malware's perspective. In this paper, we flip this conventional paradigm and discuss anti-emulation techniques in terms of *protecting Commercial-Off-the-Shelf (COTS) software*. Due to the higher requirements for usability, existing anti-emulation techniques are inapt for large-scale application vendors. To overcome such problem, we introduce three new techniques in vendors perspective for deploying their product. We evaluate the efficacy of our techniques in five aspects: (i) fast detection speed, (ii) high accuracy, (iii) low power consumption, (iv) a broad range of compatibility, and (v) high cost of bypassing. Based on our experiments, we demonstrate that misaligning the vectorization (e.g., Intel SIMD, ARM NEON) can be utilized as a promising anti-emulation technique among the proposed ones. To confirm the effectiveness, we applied our technology against 176 real Android devices and various emulators as a test bed.

1. Introduction

Runtime detection of the software running environment is a well-known topic in the field of reverse-engineering and malware analysis, and so forth. However, this topic is usually focused on malware perspective. There are many malware analysis frameworks based on code emulation. To hinder such analysis, malware use anti-emulation techniques and hide their runtime behavior to protect themselves from being

[*] Corresponding author.
E-mail addresses: daehee87@kaist.ac.kr (D. Jang), yunjong@kaist.ac.kr (Y. Jeong), sungman@kaist.ac.kr (S. Lee), dinggul@kaist.ac.kr (M. Park), kh243.kwak@samsung.com (K. Kwak), donguk14.kim@samsung.com (D. Kim), brentkang@kaist.ac.kr (B.B. Kang).

https://doi.org/10.1016/j.cose.2019.02.005
0167-4048/© 2019 Elsevier Ltd. All rights reserved.

emulated. To this end, various methods for anti-emulation technique has been explored in both academia and industry. In general, the efficacy of existing anti-emulation techniques is well suited for malware.

Recently, however, commercial application vendors are also using anti-emulation techniques on top of obfuscation in order to fortify their product from being analyzed. From a corporate perspective, the anti-emulation technique can be utilized for: (i) securing the intellectual property from emulation-assisted reverse-engineering, and (ii) preventing piracy of using the application service without purchasing the actual hardware product. The problem of using emulation techniques for stealing Intellectual Property (IP) and piracy has been pointed out previously (Collberg, 2011; Conley et al., 2003). This indicates that legitimate software vendors also require anti-emulation techniques for protection purpose.

Unfortunately, existing anti-emulation techniques are inapt for large-scale application vendors. For example, heuristic-based anti-emulation techniques such as “checking the CPU name, process list, or file-system artifacts” are inaccurate and trivially bypassed. Checking a specific emulator implementation bug can be unreliable depending on the exact emulator version. Checking the elapsed CPU clocks while executing some instructions are often unreliable due to external interrupts. In academic literature, the following techniques have been so far introduced: (i) aggregating various heuristics and statistical information for emulator detection (Vidas and Christin, 2014; Jing and Hu, 2014), (ii) using relative/absolute timing discrepancies of the kernel-level instructions that involve architecture specific features (Garfinkel et al., 2007; Raffetseder et al., 2007). All such previous detection techniques are suited for malware detecting the emulated environment; however, unacceptable for protection feature for commercial software deployment due to performance and accuracy.

In this paper, we flip the conventional view of the emulator detection research and focus in terms of software vendors protecting their application from dynamic analysis and piracy. Anti-emulation technique, in this case, should satisfy five requirements: (i) technique should not notably harm the service performance, (ii) technique should be accurate to prevent service failure, (iii) technique should not involve too much power consumption, (iv) technique should be compatible with various system configurations, and (v) technique should not be easily bypassed by attackers who intends to reverse-engineer the product.

Discovering new anti-emulation technique is not the main contribution part of this paper as there are many existing methods previously discovered. However, making a commercially-deployable anti-emulation technique is a non-trivial research issue. The emulator detection techniques introduced in this paper avoid heuristics and leverage CPU architecture specific features. We first introduce three detection techniques based on (i): context-switch granularity, (ii) translation caching of guest basic-block, and (iii) deliberate use of unaligned memory access with vectorization instructions and evaluate if they are suited for our purpose. Among the three techniques, we find that *misaligned vectorization* based detection technique most promising (although not perfect) for our purpose. All the techniques in this paper do not require any, kernel-level privilege, and it is implemented as an Android JNI library as a prototype of Samsung’s next-generation mobile security feature. Another significant advantage of our technique is that it does not require any sensitive Android permissions such as `LOCATION` or `SENSORS` to determine the environment. Requesting users for such permissions can be considered dangerous due to a security breach and privacy issues. To confirm the reliability and deployability of the proposed technique, we applied it using 176 Android mobile devices of various device vendors and several emulators. The contribution of this paper can be summarized as follows:

- This is the first paper that explores the efficacy of anti-emulation techniques regarding legitimate software vendors rather than malware.
- We introduce and evaluate three emulator detection techniques which leverage CPU architecture specifics and emulator internals thereby not depending on Android-specific features.
- We implemented commercially deployable anti-emulation technique as JNI library and tested against 176 different Android mobile devices and several emulators.

2. Background

2.1. *Emulation engines*

In this paper, we assume the QEMU (Bellard., 2005) as de-facto standard emulation engine. Most of the emulation based frameworks (e.g., Anubis (anu), Android Virtual Device) are based on QEMU. However, we also consider our anti-emulation techniques with other engines as well. Unicorn (Quynh and Vu) is a new emulation engine that specifically focuses to various CPU emulation (Intel, ARM, MIPS, and so forth) and implements additional features on top of QEMU. Bosch is a different branch of emulation engine which focuses on supporting full system emulation of Intel architecture. Considering that our technique is being developed for mobile devices, we are more interested in ARM architecture emulation.

2.2. *Previously known techniques*

There are a number of previously known techniques to distinguish the QEMU execution environment from real hardware. One of the simplest ways is to look up various names inside the execution environment. The names could be the CPU name, device driver name, etc. For instance, QEMU by default uses its unique CPU name and a device driver name such as `QEMU CPU` or `QEMU HARDDISK`. This approach seems unsatisfactory, yet it is effective against naïve system emulators, thus it is commonly observed in practice. A timing attack is another well-known technique for QEMU detection. The simplest and most typical example is to measure the consumed CPU clock cycle count of an instruction. When an instruction is emulated, the consumed CPU clock cycle count for executing the emulated instruction usually becomes higher than it should be in a real CPU. The reason is that the emulator translates the single guest instruction into a set of multiple emulator-generated host instructions.[1] If the cycle count exceeds a predefined normal range, the malware assumes that the execution environment is not real. Asides from the discussed detection techniques, there are also numerous additional emulator detection techniques proposed in academia and industry (Strazzere, 2013; Vidas and Christin, 2014; fra; Alzaylaee et al., 2017; Lee, 2014).

2.3. *QEMU internals*

Some of the detection techniques we discuss here take advantage of the architectural design of the emulation component known as the Tiny Code Generator (TCG). TCG is one of the core components in QEMU. The main role of TCG is to convert the guest instruction set (which is not executed with a physical CPU) into multiple host instruction sets (which are executed with a physical CPU) that update the emulated machine state to achieve a semantically equivalent result. The TCG performs this job in a *basic block granularity*.[2] The translated basic block is executed by a physical CPU. This indicates that once QEMU executes the converted basic-block, other QEMU emulation components (i.e., interrupt handling) will not be processed until the TCG finishes processing the converted instructions of the basic block. This behavior can be observed from the QEMU source file “target-i386/translate.c” in the case of x86 architecture. In general, QEMU processes the required subroutines for emulation in sequential order.

2.4. *Translation Block Cache*

The QEMU (and emulators in general) adopts the concept of translation caching a so-called *Translation Block Cache (TB-cache)* that significantly enhances the emulation performance. Each time when QEMU translates a particular basic block of the guest program, the translated basic block (TB) is cached to avoid repeating the same translation process. If the execution flow of the program inside QEMU encounters the same guest basic block, the translation process can be omitted, and instead, the translated code from the TB cache can be executed. This caching mechanism significantly improves the performance of QEMU.

2.5. *Vectorization*

Vectorization is a CPU technology that supports calculating multiple data with a single instruction. For example, Intel supports various “Single Instruction, Multiple Data (SIMD)” instruction set such as `MOVNTPS, MOVAPS` to support vectorization. Similarly, ARM supports NEON instruction set such as `VLDMIA, VPUSH` and so on. The main purpose of vectorization instruction is to improve the performance of the multimedia application which performs extensive graphics rendering. Here, we deliberately use such instructions in a prohibited way to detect emulated software environment.

3. Design

In this section, we introduce three emulator detection techniques that are based on the architectural design of hardware/emulator namely: (i) context-switch based detection, (ii) TB-cache based detection, and (iii) misaligned vectorization based detection. Among the detection methods, the alignment-based technique outperformed others in many aspects; therefore it is suited for commercial application vendors for protecting their application from malicious reverse-engineering to crack the software product. After discussing the design efficacy of each techniques, we show their proof-of-concept implementation and evaluation respectively then focus to the third technique (misaligned vectorization) and show our full implementation which we refer as `isEmu`. `isEmu` is compared with existing heuristic-based detection techniques shown in cal.

3.1. *Context switch based detection*

The context switch-based QEMU detection technique leverages the race condition between two threads lacking proper locking mechanism. The technique does not require any kernel privilege nor depends on timing discrepancy. In multi-threaded programming, multiple threads are executed together by sharing the CPU time slice given by the scheduler. This is possible because of the *context switch* support from the hardware and OS. In general, an involuntary context switch occurs when an external timer event interrupts the CPU. Based on the context switching and the QEMU interrupt handling mechanism, the following facts can be observed; thus we can design a method for distinguishing the QEMU environment accordingly.

- QEMU uses basic-block granularity to translate and execute its guest code.
- QEMU does not process an external interrupt while a basic-block of guest code is being executed.

As mentioned in the background section of this paper, context switching never happens in the QEMU environment while a basic block is being executed. However, this behavior is not observed in the real CPU environment. In a nutshell, *the instructions inside a basic block are executed atomically inside the QEMU environment, whereas no such atomicity is observed with real hardware*. Using this feature, we can distinguish the QEMU and real CPU by deliberately running a multi-threaded code that has a race condition problem. We can write such code by deliberately not using a lock[3] for a critical section consisted of a single basic block. By running this code, we can easily reach the race condition state in a real CPU environment; however, the race condition never occurs in a QEMU environment. Fig. 1 demonstrates this situation in detail.

In Fig. 1, a globally shared variable `N` is increased by one and decreased by one inside a critical section consisting of a single basic block. If threads enter this critical section one at a time, we would never observe the `N` becoming larger than 1.

[1] In general, `guest` indicates an emulated system and `host` indicates a real hardware-based system is running emulation software.

[2] Basic block is a set of instructions with no branch executed in sequential order.

[3] A lock could be a spinlock, mutex, semaphore, and so forth.

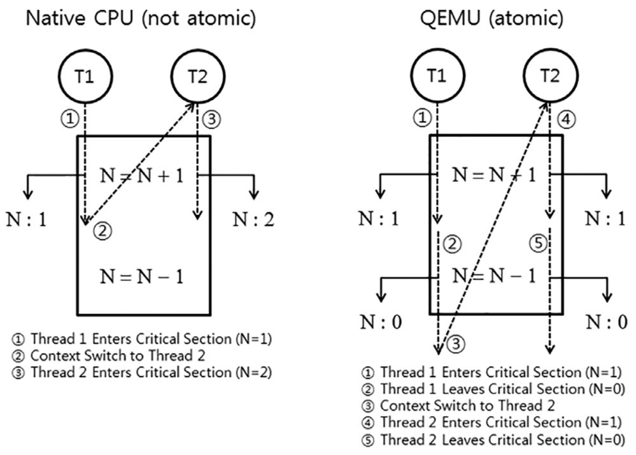

Native CPU (not atomic)
QEMU (atomic)
T1
T2
①
②
③
N = N + 1
N = N − 1
N : 1
N : 2
① Thread 1 Enters Critical Section (N=1)
② Context Switch to Thread 2
③ Thread 2 Enters Critical Section (N=2)
T1
T2
①
②
③
④
⑤
N = N + 1
N = N − 1
N : 1
N : 1
N : 0
N : 0
① Thread 1 Enters Critical Section (N=1)
② Thread 1 Leaves Critical Section (N=0)
③ Context Switch to Thread 2
④ Thread 2 Enters Critical Section (N=1)
⑤ Thread 2 Leaves Critical Section (N=0)

Fig. 1 – Different context switching behavior between real CPU and QEMU. A circle denotes a thread, and a box denotes a critical section composed of a single basic block.

However, if multiple threads were about to enter this critical section together at the same time, we could find the value of N being larger than that of 2. As we discussed above, a guest context switch event does not occur during basic block execution. As a result, QEMU happens to protect the critical section (consisting of a single basic block) even if the program does not use a proper locking mechanism such as *mutex* or *spinlock*. However, in a real CPU environment, we can observe N becoming larger than two, which is an abnormal race condition state.

Therefore, it can be utilized as a universal emulation detection technique that can be used regardless of system configuration specifics. Unfortunately, this detection technique requires sufficiently long time (e.g., seconds) to accurately conclude the running environment (if things are fortunate, race condition state could not occur in real hardware for a long time). Also, as the technique requires exhaustive repetition of loops, it requires high power consumption. In terms of bypassing the anti-emulation, we consider it is safe from trivial bypassing because this phenomenon does not involve any architectural specifics and it is inevitable to avoid unless the TCG implementation logic translates the guest code into a *single instruction granularity*. However, such implementation logic severely degrades the overall emulation performance. Finally, this anti-emulation is only plausible for detecting full-system emulators that emulate external hardware signals, thus cannot be applied to detect Unicorn engine.

We made proof-of-concept C code and verified it works successfully against three system emulators (`qemu-system-i386/x86_64/arm`) and real CPUs. Using such code, we later show detailed experiment results of this technique in Section 4 considering the number of CPU cores and size of the unlocked basic-block.

3.2. *Translation Block Cache based detection*

Most of the dynamic binary instrumentation (DBI)-based emulators, including QEMU, use a code translation caching mechanism to accelerate the performance (details are explained in the background section). Although this caching system speeds up the emulation performance, it creates a significant timing difference against a real CPU, thus can be utilized for detecting the emulated environment.

Using this additional layer of the caching system enables us to create a significant timing discrepancy regarding *self-modifying code* that overwrites itself during its execution. In QEMU, overwriting the code memory breaks the cache coherence of the TB cache. Therefore, the *self-modifying code* cannot leverage the TB cache system of QEMU. When a self-modifying code runs inside the QEMU environment, the TB cache loses its effectiveness and severely degrades the performance of code execution speed compared to the non-self-modifying code.

Similarly to the TB cache of emulators, the hardware cache (e.g., L1) also experiences degraded performance when a code invalidates itself. However, the impact thereof on the performance is trivial relative to TB cache invalidation. With high assurance, we can conclude that if the performance of self-modifying code dramatically deteriorates (i.e., by orders of magnitude) compared to other code, we can reasonably suspect that the running environment is being emulated.

An example of detection can be performed as follows: Consider two code snippets A: a code fragment consisting of 1000 lines of ordinary (non-self-modifying) assembly instructions such as `mov`, `add`, and `push`; and B: a self-modifying code fragment consisting of 100 lines of the same set of assembly instructions. If we were to iterate the code execution of A and B numerous times in a real CPU, the total iteration time of A would be expected to exceed that of B because there are orders of magnitude more instructions to execute. However, in an emulation environment, this timing experiment would show the opposite result since the TB cache would not be utilized in B (self-modifying code) after all.

The advantages of this technique are that it does not require kernel-level privilege and it is architecture agnostic. Also, mitigating this detection attempt is practically impossible as the overall emulation performance is significantly affected. However, this technique is inadequate to apply to commercial software because the detection requires a lot of time for an accurate result, and still, the accuracy cannot be guaranteed regardless of the cache performance and other timing issues. For example, the elapsed time of any code execution could exceptionally be changed due to unexpected asynchronous hardware interrupts. Also, measuring the detailed timing requires exhaustive code execution thus consume power. In terms of bypassing the anti-emulation, it could be trivially bypassed by disabling the QEMU to use translation caching. However, the overall performance severely drops without translation caching feature.

We made a proof-of-concept detection code in C and verified it works successfully against several system emulators `qemu-system-i386/x86_64/arm, unicorn` and several Intel CPUs. In the case of ARM architecture, this technique has another issue regarding the i-cache/d-cache separation. Due to the implementation of the caching system, ARM architecture can execute the code before it was invalidated by self-modification. Because of this reason, self-modifying code in ARM architecture shows unexpected behavior unless the cache is handled correctly. We can take advantage of this feature as an indicator of distinguishing the real ARM hardware from the emulator.

3.3. *Misaligned vectorization based detection*

Memory access alignment issue stems from the incapability of earlier (and current) CPUs accessing the cache with byte-granularity. For example, some 32-bit CPU architectures have a 30-bit addressing line for fetching the memory. Due to the lack of 2 bits, such CPU can access the memory only if the target memory address is multiple of 4 (`100 in binary`). For such reason, if the CPU wants to access a memory address that is not the multiple of 4, the CPU fetches memory twice and re-assemble the memory contents. Vectorization instructions of Intel and ARM do not support unaligned memory access at the hardware level (even with kernel modification for handling alignment feature) thus raises an application-aware fault. In general, the kernel is capable of handling unaligned memory access even if the CPU is incapable of handling such type of memory access. For example, Linux kernel provides an interface for enabling/disabling faulting behavior upon unaligned access in general purpose instructions.

However, high-performance vectorization operations such as Intel SIMD and ARM NEON are guaranteed to raise fault upon unaligned access regardless of kernel configuration. If such fault occurs, the user application is notified by kernel since CPU is incapable of proceeding execution. However, software emulators do not have to suffer from such issues because any memory access is ultimately reconstructed with multiple combinations of operations at a software level. As a result, unaligned vectorization inside emulator executes the correct semantic of unaligned vectorization with similar performance compared to the aligned vectorization. Here, we take advantage of this phenomenon to effectively distinguish the software-emulated environment and native hardware environment without any kernel-dependency.

The algorithm design of our detection is as follow: First, we install a fault-handler to catch the hardware fault signal induced by unaligned vectorization. Once the handler is installed, the actual detection routine deliberately makes an unaligned pointer and dereference such pointer with vectorization instructions that are affected by target memory alignment (e.g., `MOVNTPS` of Intel, `VLDMIA` of ARM). If the running environment is emulated, nothing happens upon such memory access. However, in the real hardware-based environment, CPU immediately raises the fault signal and invokes our callback handler. Based on such different behavior, we decide if the running environment is emulated or not. Fig. 2 is the overall process depicting our detection technique based on misaligned vectorization. Once, the signal handler setup process is done while application loading, detection process (dereferencing the unaligned pointer with specific instruction) is invoked unpredictably.

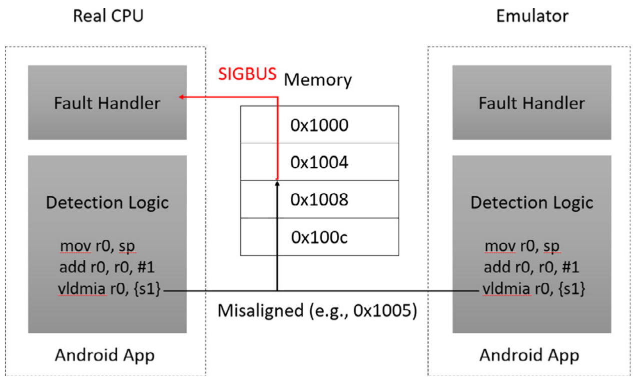


Fig. 2 – Different result for misaligned vectorization in real CPU and emulator. The alignment fault occurs if CPU is real hardware regardless of kernel. Emulators process unaligned vectorization as it does not break the correctness of the system.

Overall, this technique is sufficiently fast and accurate, plus it does not use a lot of CPU cycles thus power consumption is low. Also, it has high compatibility with many devices with various configuration. We applied this technique to 176 real Android devices and QEMU-based emulators and Unicorn, Bosch engine. Unfortunately, there are few extraordinary cases among devices that we tested[4]; and Bosch engine does implement the misalignment handling for vectorization thus raises faults upon the emulated unaligned memory access. Still, we believe that this is so far the most promising technique for anti-emulation for commercial application vendors to provide reasonable performance and service quality with a required level of security.

4. Implementation and evaluation

In this section, we first discuss the implementation and evaluation of the two aforementioned anti-emulation techniques (context-switch, translation cache). Afterward, we show the implementation of `isEmu` which utilizes the third technique (misaligned vectorization, which we find promising) and conduct detailed evaluation comparing with previously existing techniques. We provide a private link for the full source code of `isEmu`.[5,6] We also attached proof-of-concept codes in the appendix including other techniques.

4.1. *Context switch based detection*

We implemented the context-switch based emulator detection technique by writing a multithreaded application that deliberately unlocks the critical section. In a global memory area, we declare a simple counter variable (initialized to zero) that is shared among two threads. The thread repeats a simple operation of incrementing (zero to one) the shared variable and decrementing (one to zero) it. It is important that the codes for the operation are gathered inside one single basic block. Because locks do not protect the critical section that accesses the shared-variable, a proper race-condition can temporarily

[4] Due to some exceptional Intel-based Android devices using binary-translation with `libhoudini`.

[5] https://www.dropbox.com/s/8atpfzkv6wu160u/isemu.zip?dl=0.

[6] Due to corporate policy; we would like to ask reviewers and editor to keep the link private.

cause shared variable state for *two*. However, since the emulator process context-switch only between basic block translation, no context-switch occurs within a basic block; thus the race condition never happens if emulated. Algorithm 1 is the pseudo-code for implementation.

Algorithm 1 Context switching-based detection.

```
Data: boolean isEmu, integer N, function AtomicThread
Result: the final value of isEmu
isEmu = true
N = 0
AtomicThread = (Repeat two operations atomically: N=N+1, N=N-1)
BeginThread( AtomicThread )
BeginThread( AtomicThread )
while N less than 2 do
    if Detection time out then
        return isEmu
    else

    end
end
isEmu = false;
return isEmu
```

The big question of context-switch based emulator detection is *how quickly and reliably the race condition occurs in real-hardware?*. Based on the design of detection algorithm, we believe there are two major factors that affects the speed and reliability of this technique: (i) number of CPU cores, and (ii) size of critical section block. To evaluate the efficacy of the algorithm, we conducted two experiments regarding this issue.

The first experiment is measuring the minimal loop counts until the race-condition state (N=2) is observed. We implemented the algorithm Algorithm 1 with C and compiled into Intel and ARM binary. Then counted the loop counts with various configuration of CPU core numbers.[7] Fig. 3 shows the result of this experiment. From the graphs, X axis is the number of iteration for escaping the loop and the Y axis number is the repeated number of evaluation trials for the same value of X (block size of critical section is 1024 bytes). From the evaluation result, we can see that this technique is unreliable in case the CPU core is single (takes too much time to observe the race condition). The race-condition state is quickly reached in case the CPU has multi-core. However, if the number of CPU core is more than four, there was no particular difference.

The next experiment is to measure how the size of critical section affects this technique. Obviously, if the size of critical section is large, the chance of context switch between the increment and decrement will increase, thus affect the probability of observing the race condition state. Since the single-core environment shows the worst performance for detection, we tried increasing the size of critical section and wondered if it will increase the detection performance in single-core environment. Fig. 4 shows our evaluation result. As expected, increasing the distance between increment operation and decrement operation affected the performance of context-switch based detection.

4.2. *Translation cache based detection*

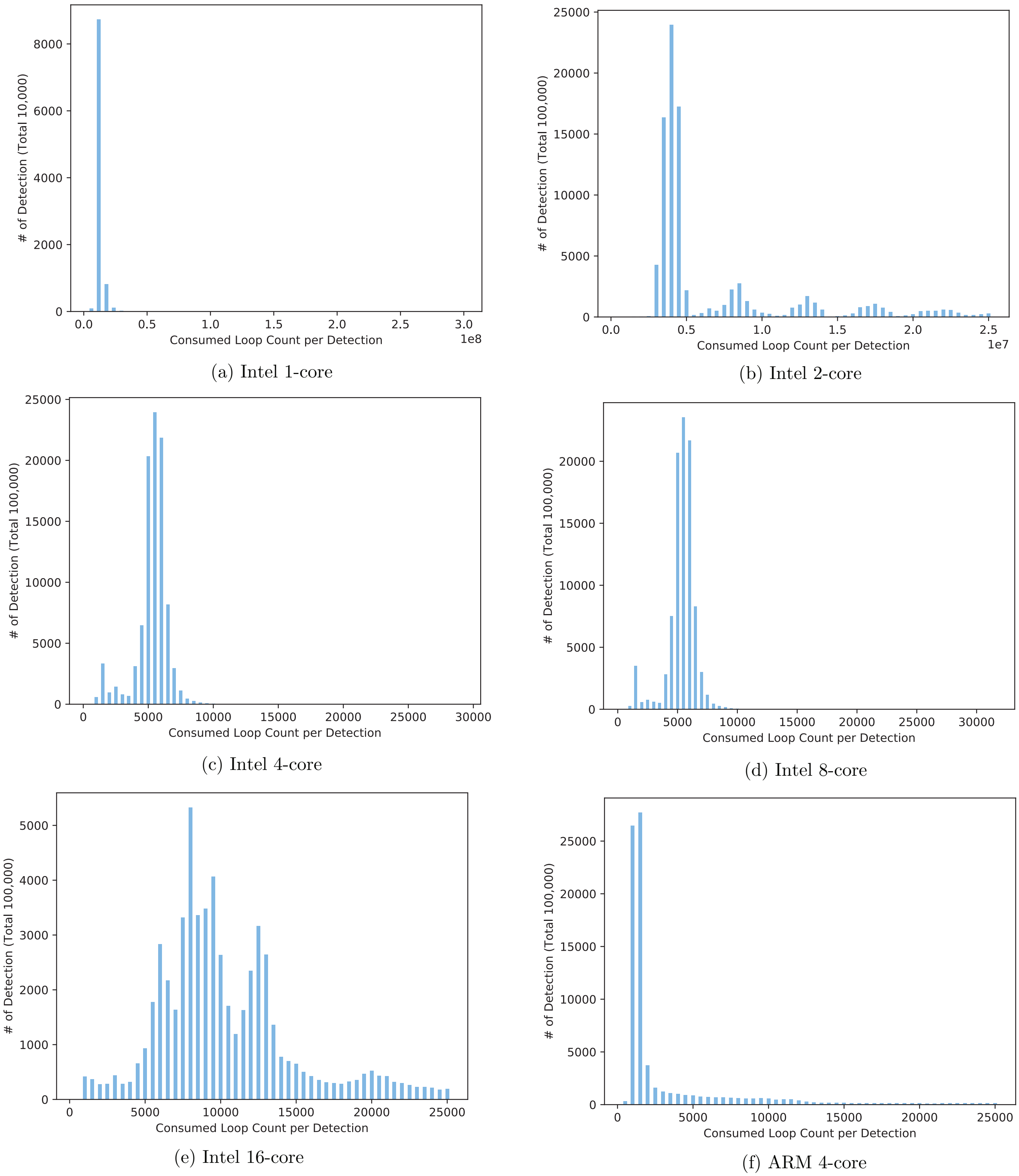
Translation cache is an essential part of general emulators. Iterating same translation for same guest code significantly slow down the overall performance of emulation. Therefore, emulator caches the result of translation and re-use if the guest code execution encounters the same code. The detection using this feature is based on timing. Our proof-of-concept implementation measures the timing between self-modifying code and regular codes. The self-modifying code is implemented by allocating a readable-writable-executable memory at a fixed memory address and placing a code that overwrites the memory content of its code address itself. The regular code is simple no-operation codes which only consumes CPU cycles. Because the self-modifying code invalidates the translation cache, the performance impact is significant. By comparing the relative performance ratio between the self-modifying code and regular codes, we can deduce the running environment with high confidence. However, any timing based detection approach cannot guarantee its result. Algorithm 2 is the pseudo-code for implementation and Table 1 is the evaluation result.

Algorithm 2 Translation cache-based detection.

```
Data: boolean isEmu, time Tstart, time Tend, time D1, time D2
        function selfMod, function dummyFunc
Result: the final value of isEmu
selfMod = (Function modifies its code during execution)
dummyFunc = (Function executes dummy instructions)
Tstart = time() Execute selfMod Tend = time()
D1 = Tend - Tstart
Tstart = time() Execute dummyFunc 100 times Tend = time()
D2 = Tend - Tstart
if D1 greater than D2 then
    isEmu = true
else
    isEmu = false
end
return isEmu
```

The evaluation suggests that the performance impact of invalidating code cache is significantly higher than real hardware. In case of the actual device, standard code that has a sufficiently higher number of instruction took longer execution time than short self-modifying code. However, in case of the emulator, self-modifying code shows orders of magnitude slower performance. In case of ARM real environment, executing self-modifying code requires special cache handling, unless the code shows unexpected behavior. In ARM architecture, the instruction cache (i-cache) and data cache (d-cache) is separated and has loose coherency. If the code updates itself with data access instructions, the code in memory is updated however the instruction cache remains valid. Therefore, the execution after self-modification could execute the code before it is supposedly updated, which causes unexpected behavior.

[7] We control number of cores via Linux CPU Pinning support with `taskset` utility.



(a) Intel 1-core

(b) Intel 2-core

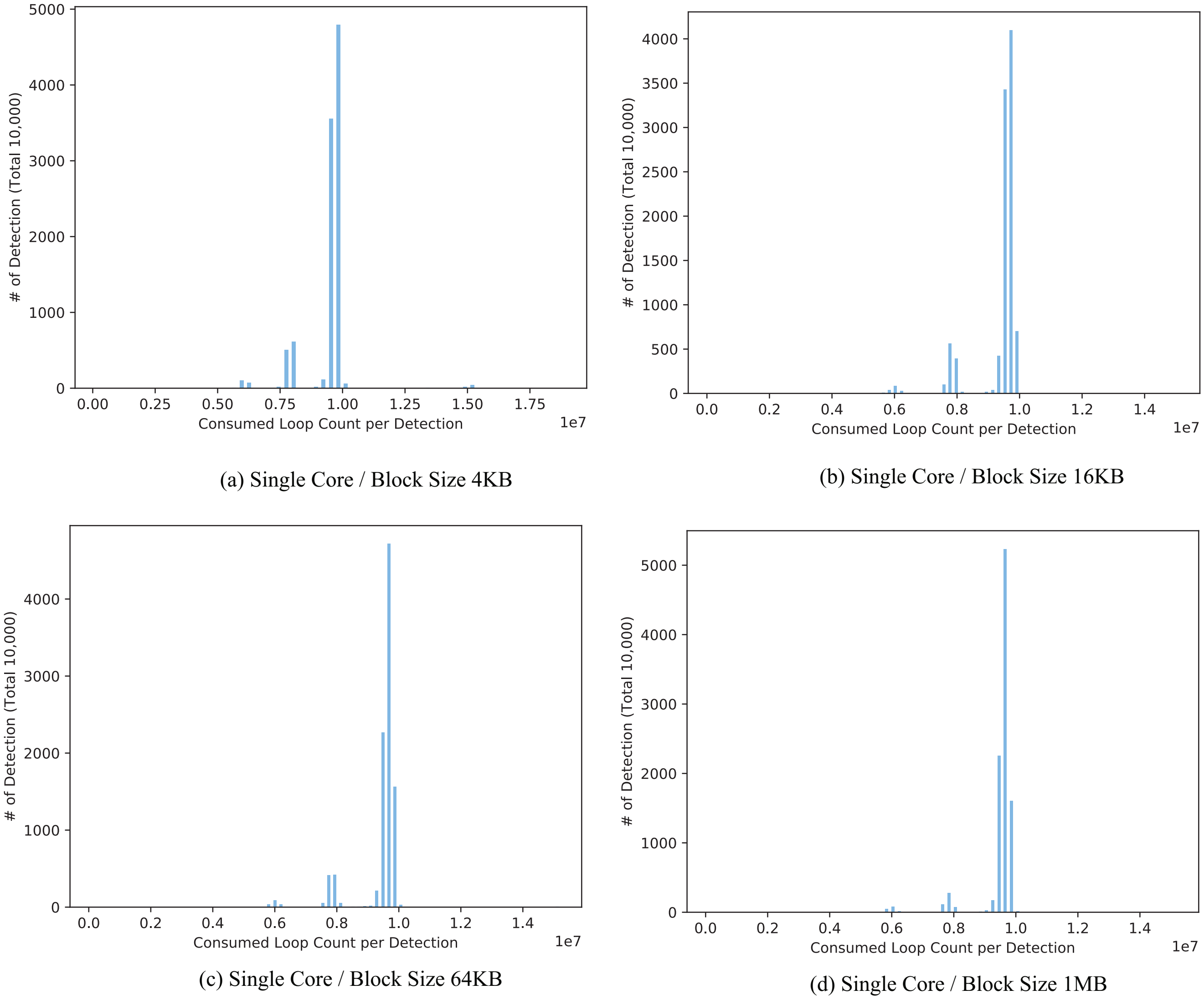
(c) Intel 4-core

(d) Intel 8-core

(e) Intel 16-core

(f) ARM 4-core

Fig. 3 – Number of loop counts before observing the race-condition state in real hardware. X-axis is the number of iteration for escaping the main loop (from pseudo-code) and the Y-axis is the repeated number of evaluation trials for the same X-axis result. Block size of critical section is 1KB.



(a) Single Core / Block Size 4KB

(b) Single Core / Block Size 16KB

(c) Single Core / Block Size 64KB

(d) Single Core / Block Size 1MB

Fig. 4 – Relationship between critical section block size and efficacy of context-switch based technique. Experiment configurations are same as Fig. 3.

Regarding speed, the detection was faster than we initially expected because the relative timing difference between standard code and self-modifying code was high. However, occasionally the timing shows high variance due to external hardware interrupts, which makes this technique less reliable in various hardware environments.

4.3. *isEmu: unaligned vectorization based detection*

The main goal of this research is finding a sufficiently accurate, compatible and fast emulator detection technique for commercial deployment. Based on our study, we develop `isEmu`: a fast, accurate Android anti-emulation technique. Considering commercial deployment, the detection attempt should catch emulators, but more importantly, should not interfere benign users using real devices. To demonstrate the improvement of `isEmu` in this perspective over existing techniques, we use a publicly available Android emulator detection tool (cal) for comparison. This tool aggregates various emulator detection heuristics that are previously discussed (Vidas and Christin, 2014).

`isEmu` is implemented as JNI library to target ARM Android devices. The JNI code initially setup signal handler at application load time. Once the handler is initialized, the application can detect the running environment. To install a signal handler for catching alignment-fault (SIGBUS of Linux), we use standard `sigaction` GLIBC API and register our callback function. The time consumption of this process is negligible as additionally calling one more GLIBC API upon process loading. After the signal handler is installed, the actual detection code makes an unaligned pointer (e.g., a 32-bit width pointer that points valid memory address but the address is not multiple of 32-bit such as `0x10033`). After making such a pointer, use `LDM/STM` instruction to dereference such unaligned pointer.

Table 1 – Timing discrepency of code cache invalidation in hardware and emulator. The execution time is the sum of 100,000 iteration. Self-Mod is composed with single memory access code that updates itself. Normal is composed with addition loop with 100 iteration.

CPU model	Type of code	Execution time (ms)	Remarks
ARMv7	Self-Mod	N/A	SIGSEGV (due to cache issue)
ARMv7	Normal	13	
Intel e5-2630 (32bit)	Self-Mod	10	
Intel e5-2630 (32bit)	Normal	25	
Intel e5-2630 (64bit)	Self-Mod	10	
Intel e5-2630 (64bit)	Normal	25	
Intel i5-4670 (32bit)	Self-Mod	15	
Intel i5-4670 (32bit)	Normal	22	
Intel i5-4670 (64bit)	Self-Mod	13	
Intel i5-4670 (64bit)	Normal	23	
Intel i7-6700 (32bit)	Self-Mod	17	
Intel i7-6700 (32bit)	Normal	21	
Intel i7-6700 (64bit)	Self-Mod	11	
Intel i7-6700 (64bit)	Normal	22	
QEMU-ARMv7	Self-Mod	700	ARM mode
QEMU-ARMv7	Normal	137	ARM mode
QEMU-i386	Self-Mod	620	
QEMU-i386	Normal	38	
QEMU-x86_64	Self-Mod	631	
QEMU-x86_64	Normal	40	
Unicorn-ARM	Self-Mod	150	ARM mode
Unicorn-ARM	Normal	50	ARM mode

According to our experiments, ARM instructions such as LDR/STR supports hardware-level unaligned access since ARMv6. However, an instruction such as LDM/STM lacks the capability of dereferencing unaligned pointers in general ARM architecture including the latest. We also implemented Intel version of emulation detector using SSE instructions. The overall mechanism of detection technique is the same as the previously explained way (JNI implementation for ARM Android device). The key difference between Intel emulator detector and ARM emulator detector is that Intel version uses MOVNTPS SSE instruction instead of LDM/STM of the ARM. Similarly to ARM case, Intel MOVNTPS (and similar family of other SSE instructions and vector instructions) requires specific address alignment (16 for MOVNTPS) while accessing memory. Algorithm 3 is the pseudo-code for implementation.

Algorithm 3 Unaligned vectorization-based detection.

```
Data: boolean isEmu, pointer PTRfunction AlignTrapHandler
Result: the final value of isEmu
isEmu = true
AlignTrapHandler = (change isEmu to false)
InstallHandler(AlignTrapHandler)
PTR = misaligned pointer for Read/Write memory
Execute Vectorization with PTR
return isEmu
```

To evaluate the performance impact of isEmu, we ran microbenchmark to measure the consumed clock cycles for installing the signal handler and processing time of the hardware signal due to unaligned memory access. To evaluate the accuracy, we applied isEmu to test Android application and ran inside 176 real ARM Android devices[8] and several emulators including qemu-system-arm and standard Android AVD emulator. We also ran the previously existing Android emulator detection tool (cal). This tool executes several Android emulator detection heuristic codes written in Java and combines its result with bitwise OR (0: real device, 1: emulator). We dissected the code into three heuristics as follow.

- H1. Android API based artifacts.
- H2. Emulator property artifacts.
- H3. Filesystem artifacts.

We measured the code execution speed and accuracy of each heuristics and summarized its comparison result with isEmu. Three tables Tables 2–4 (for convenience, we split the evaluation result into three tables) summarizes the evaluation. In the table, ✔indicates that the detection code reported the accurate result as intended. ✖indicates that the detection code reported the opposite result. ✱indicates the detection code was not compatible and refused to run. The evaluation initially used 176 Android devices (for isEmu) provided by Amazon Device Farm (Android-All option). However, due to frequent device updates, some of the devices were removed. In such case, we makred the evaluation result with "-" symbol. The evaluation shows that isEmu shows 100% accurate result for all Device Farm cases and emulators.[9] However, the

[8] We use Amazon Device Farm for this experiment (Amazon, 2006).

[9] isEmu reports x86 real Android devices as “Emulator”. This could be considered the wrong result, but we claim that this is technically not a false positive as such real devices indeed used emulation technique for code execution. However, to distinguish

Table 2 – Deployment test summary (1/3).

Device	OS	Vendor	Arch	isEmu	H1	H2	H3
Nexus 7 – 1st (WiFi)	4.2.1	ASUS	armeabi-v7a	✔	✖	✖	✖
Nexus 7 – 1st (WiFi)	4.2.0	ASUS	armeabi-v7a	✔	✖	✖	✖
Nexus 7 – 1st (WiFi)	4.3.0	ASUS	armeabi-v7a	✔	✔	✖	✔
Nexus 7 – 1st (WiFi)	4.4.2	ASUS	armeabi-v7a	✔	✖	✖	✔
Nexus 7 – 2nd (WiFi)	5.0.1	ASUS	armeabi-v7a	✔	✖	✖	✔
Nexus 7 – 2nd (WiFi)	4.4.2	ASUS	armeabi-v7a	✔	✖	✖	✔
Nexus 7 – 2nd (WiFi)	4.4.4	ASUS	armeabi-v7a	✔	✖	✖	✔
Nexus 7 – 2nd (WiFi)	4.3.1	ASUS	armeabi-v7a	✔	✔	✖	✔
Nexus 7 – 2nd (WiFi)	6.0.0	ASUS	armeabi-v7a	✔	✖	✖	✔
Memo Pad 7	4.4.2	ASUS	x86	✔	✖	✔	✔
Fire HD 7 (2014)	4.4.3	Amazon	armeabi-v7a	✔	✖	✔	✔
Kindle Fire HDX 7	4.4.3	Amazon	armeabi-v7a	✔	✖	✔	✔
Blackberry Priv	5.1.1	Blackberry	arm64-v8a	✔	-	-	-
Galaxy S8 Unlocked	8.0.0	Samsung	arm64-v8a	✔	✔	✔	✔
Galaxy Tab 3 10.1	4.2.2	Samsung	x86	✔	✷	✷	✷
Pixel	8.0.0	Google	arm64-v8a	✔	✖	✖	✔
Pixel	7.1.2	Google	arm64-v8a	✔	✖	✖	✔
Pixel 2	8.1.0	Google	arm64-v8a	✔	✖	✖	✔
Pixel 2	8.0.0	Google	arm64-v8a	✔	✖	✖	✔
Pixel 2 XL	8.0.0	Google	arm64-v8a	✔	✖	✖	✔
Pixel XL	7.1.2	Google	arm64-v8a	✔	✖	✖	✔
Pixel XL	8.0.0	Google	arm64-v8a	✔	✖	✖	✔
Desire 526G+	4.4.2	HTC	armeabi-v7a	✔	–	–	–
One A9 (Unlocked)	6.0.1	HTC	arm64-v8a	✔	✷	✷	✷
One M7 (AT/T)	4.4.2	HTC	armeabi-v7a	✔	✖	✔	✔
One M8 (AT/T)	4.4.4	HTC	armeabi-v7a	✔	✖	✔	✔
One M8 (AT/T)	4.4.2	HTC	armeabi-v7a	✔	✖	✔	✔
One M8 (Verizon)	4.4.2	HTC	armeabi-v7a	✔	✖	✔	✔
One M8 (Verizon)	4.4.4	HTC	armeabi-v7a	✔	✖	✔	✔
One M9 (AT/T)	5.0.2	HTC	arm64-v8a	✔	✖	✔	✖
One M9 (Verizon)	5.0.2	HTC	arm64-v8a	✔	✖	✔	✖
Ascend Mate 7	4.4.2	Huawei	armeabi-v7a	✔	✔	✔	✔
Honor 6	4.4.2	Huawei	armeabi-v7a	✔	-	-	-
M8	6.0.0	Huawei	arm64-v8a	✔	✖	✔	✔
P9	6.0.0	Huawei	arm64-v8a	✔	✖	✔	✔
Aqua Y2 Pro	4.4.2	Intex	armeabi-v7a	✔	✔	✖	✔
G Flex (AT/T)	4.2.2	LG	armeabi-v7a	✔	✔	✔	✖
G Pad 7.0 (AT/T)	4.4.2	LG	armeabi-v7a	✔	✔	✔	✔
G2 (AT/T)	4.4.2	LG	armeabi-v7a	✔	✔	✔	✔
G2 (T-Mobile)	4.4.2	LG	armeabi-v7a	✔	✖	✔	✔
G3 (AT/T)	5.0.1	LG	armeabi-v7a	✔	✔	✔	✔
G3 (AT/T)	4.4.2	LG	armeabi-v7a	✔	✔	✔	✔
G3 (T-Mobile)	4.4.2	LG	armeabi-v7a	✔	✔	✔	✔
G3 (Verizon)	4.4.2	LG	armeabi-v7a	✔	✖	✔	✔
G5 (T-Mobile)	6.0.1	LG	arm64-v8a	✔	✷	✷	✷
Nexus 4	4.4.3	LG	armeabi-v7a	✔	✔	✖	✔
Nexus 5	4.4.2	LG	armeabi-v7a	✔	✖	✖	✔
Nexus 5	6.0.0	LG	armeabi-v7a	✔	✖	✖	✔
Nexus 5	5.1.1	LG	armeabi-v7a	✔	✖	✖	✔
Nexus 5	5.0.1	LG	armeabi-v7a	✔	✖	✖	✔
Optimus L70	4.4.2	LG	armeabi-v7a	✔	✖	✔	✔
V20 (AT/T)	7.0.0	LG	arm64-v8a	✔	✖	✔	✔
V20 (T-Mobile)	7.0.0	LG	arm64-v8a	✔	✖	✔	✔
V20 (Verizon)	7.0.0	LG	arm64-v8a	✔	✖	✔	✔
DROID RAZR HD	4.4.2	Motorola	armeabi-v7a	✔	✖	✔	✔
DROID RAZR M	4.4.2	Motorola	armeabi-v7a	✔	✖	✔	✔
DROID Turbo	5.1.0	Motorola	armeabi-v7a	✔	✔	✔	✔
DROID Turbo 2	5.1.1	Motorola	arm64-v8a	✔	✖	✔	✔
DROID Ultra	4.4.4	Motorola	armeabi-v7a	✔	✖	✔	✔

Table 3 – Deployment test summary (2/3).

Device	OS	Vendor	Arch	isEmu	H1	H2	H3
Moto E – 2nd	5.1.0	Motorola	armeabi-v7a	✔	✔	✔	✔
Moto E – 2nd	5.0.2	Motorola	armeabi-v7a	✔	–	–	–
Moto G (AT/T)	4.4.4	Motorola	armeabi-v7a	✔	✔	✔	✔
Moto G – 2nd	6.0.0	Motorola	armeabi-v7a	✔	✔	✔	✔
Moto G – 2nd	5.0.2	Motorola	armeabi-v7a	✔	–	–	–
Moto G – 3rd	6.0.0	Motorola	armeabi-v7a	✔	✔	✔	✔
Moto G 4	7.0.0	Motorola	armeabi-v7a	✔	✖	✔	✔
Moto X	5.1.0	Motorola	armeabi-v7a	✔	–	–	–
Moto X 2nd	5.1.0	Motorola	armeabi-v7a	✔	✖	✔	✔
Nexus 6	5.1.0	Motorola	armeabi-v7a	✔	–	–	–
Nexus 6	6.0.0	Motorola	armeabi-v7a	✔	–	–	–
Nexus 6	7.0.0	Motorola	armeabi-v7a	✔	–	–	–
Find 7a	4.3.0	Oppo	armeabi-v7a	✔	✔	✔	✔
Excite Go	4.4.2	Toshiba	x86	✔	–	–	–
Galaxy A3	5.1.1	Samsung	armeabi-v7a	✔	✖	✔	✖
Galaxy A5	5.0.2	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy E5	5.1.1	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy E7	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy Grand 2	4.4.2	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy Grand Neo Plus	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✖
Galaxy Grand Prime 4G	5.1.1	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy Grand Duos	4.4.4	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy J1 Ace	4.4.4	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy J1 Duos	4.4.4	Samsung	armeabi-v7a	✔	✖	✔	✖
Galaxy J2 4G	5.1.1	Samsung	armeabi-v7a	✔	–	–	–
Galaxy J5 4G	5.1.1	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy J7 4G	5.1.1	Samsung	armeabi-v7a	✔	✷	✷	✷
Galaxy Light	4.4.2	Samsung	armeabi-v7a	✔	✖	✔	✖
Galaxy Note 2 (AT/T)	4.4.2	Samsung	armeabi-v7a	✔	✔	✔	✖
Galaxy Note 2 (AT/T)	4.3.0	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy Note 2 (Verizon)	4.4.2	Samsung	armeabi-v7a	✔	✔	✔	✖
Galaxy Note 3 (AT/T)	4.4.2	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy Note 3 (AT/T)	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy Note 3 (Sprint)	5.0.0	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy Note 3 (T-Mobile)	5.0.0	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy Note 3 (Verizon)	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy Note 4 (AT/T)	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy Note 4 (AT/T)	5.0.1	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy Note 4 (Sprint)	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy Note 4 (T-Mobile)	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy Note 4 (Verizon)	5.0.1	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy Note 4 (Verizon)	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy Note 4 SM-N910H	5.0.1	Samsung	armeabi-v7a	✔	✖	✔	✖
Galaxy Note5 (AT/T)	5.1.1	Samsung	arm64-v8a	✔	✖	✔	✖
Galaxy Note5 (AT/T)	7.0.0	Samsung	arm64-v8a	✔	✔	✔	✖
Galaxy Note5 (T-Mobile)	5.1.1	Samsung	arm64-v8a	✔	✔	✔	✖
Galaxy Note5 SM-N920C	6.0.1	Samsung	arm64-v8a	✔	✔	✔	✖
Galaxy Note8 (Unlocked)	7.1.1	Samsung	arm64-v8a	✔	✔	✔	✔
Galaxy S DUOS 3	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S3 (AT/T)	4.3.0	Samsung	armeabi-v7a	✔	✔	✔	✖
Galaxy S3 (T-Mobile)	4.3.0	Samsung	armeabi-v7a	✔	✔	✔	✖
Galaxy S3 (Verizon)	4.3.0	Samsung	armeabi-v7a	✔	✔	✔	✖
Galaxy S3 (Verizon)	4.4.2	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S3 LTE (T-Mobile)	4.3.0	Samsung	armeabi-v7a	✔	✔	✔	✖
Galaxy S4 (AT/T)	4.4.2	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy S4 (AT/T)	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S4 (AT/T)	5.0.1	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S4 (T-Mobile)	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S4 (US Cellular)	4.4.2	Samsung	armeabi-v7a	✔	✖	✔	✔

Table 4 – Deployment test summary (3/3).

Device	OS	Vendor	Arch	isEmu	H1	H2	H3
Galaxy S4 (Verizon)	4.4.2	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S4 (Verizon)	5.0.1	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy S4 Active (AT/T)	4.4.2	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S4 Mini GT-I9195	4.4.2	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy S4 mini (Verizon)	4.4.2	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S4(Unlocked)	5.0.1	Samsung	armeabi-v7a	✔	✖	✔	✖
Galaxy S5 (AT/T)	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S5 (AT/T)	6.0.1	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S5 (AT/T)	4.4.2	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S5 (T-Mobile)	4.4.2	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S5 (Verizon)	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S5 (Verizon)	6.0.1	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy S5 Active (AT/T)	4.4.2	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy S6 (T-Mobile)	7.0.0	Samsung	arm64-v8a	✔	✖	✔	✖
Galaxy S6 (T-Mobile)	6.0.1	Samsung	arm64-v8a	✔	✔	✔	✖
Galaxy S6 (Verizon)	6.0.1	Samsung	arm64-v8a	✔	✖	✔	✖
Galaxy S6 (Verizon)	5.0.2	Samsung	arm64-v8a	✔	✖	✔	✖
Galaxy S6 Edge	5.0.2	Samsung	arm64-v8a	✔	✖	✔	✖
Galaxy S6 Edge	7.0.0	Samsung	arm64-v8a	✔	✔	✔	✖
Galaxy S6 Edge (Verizon)	5.0.2	Samsung	arm64-v8a	✔	✖	✔	✖
Galaxy S6 Edge SM-G925F	6.0.1	Samsung	arm64-v8a	✔	✖	✔	✖
Galaxy S6 Edge+ (AT/T)	5.1.1	Samsung	arm64-v8a	✔	✖	✔	✖
Galaxy S6 Edge+ (T-Mobile)	5.1.1	Samsung	arm64-v8a	✔	✔	✔	✖
Galaxy S6 SM-G920F	6.0.1	Samsung	arm64-v8a	✔	✖	✔	✖
Galaxy S7 (AT/T)	6.0.1	Samsung	arm64-v8a	✔	✔	✔	✔
Galaxy S7 (T-Mobile)	6.0.1	Samsung	arm64-v8a	✔	–	–	–
Galaxy S7 Edge (AT/T)	6.0.1	Samsung	arm64-v8a	✔	✔	✔	✔
Galaxy S7 Edge SM-G935F	6.0.1	Samsung	arm64-v8a	✔	✖	✔	✖
Galaxy S7 SM-G930F	6.0.1	Samsung	arm64-v8a	✔	✖	✔	✖
Galaxy S8 (T-Mobile)	7.0.0	Samsung	arm64-v8a	✔	✔	✔	✔
Galaxy S8+ (T-Mobile)	7.0.0	Samsung	arm64-v8a	✔	✔	✔	✔
Galaxy S9 (Unlocked)	8.0.0	Samsung	arm64-v8a	✔	✔	✔	✔
Galaxy S9+ (Unlocked)	8.0.0	Samsung	arm64-v8a	✔	✖	✔	✔
Galaxy Tab 3 7.0 (Sprint)	4.2.2	Samsung	armeabi-v7a	✔	✷	✷	✷
Galaxy Tab 3 7.0 (T-Mobile)	4.4.4	Samsung	armeabi-v7a	✔	✔	✔	✔
Galaxy Tab 3 Lite 7.0	4.2.2	Samsung	armeabi-v7a	✔	✖	✔	✖
Galaxy Tab 4 10.1 (WiFi)	5.0.2	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy Tab 4 10.1 (WiFi)	4.4.2	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy Tab 4 7.0 (WiFi)	4.4.2	Samsung	armeabi-v7a	✔	✖	✔	✔
Galaxy Tab S2 9.7	6.0.1	Samsung	arm64-v8a	✔	✖	✔	✔
Galaxy Tab S2 8.0 (WiFi)	6.0.1	Samsung	arm64-v8a	✔	✖	✔	✔
Galaxy Tab S2 8.0 (WiFi)	5.1.1	Samsung	armeabi-v7a	✔	✖	✔	✖
Venue 8 7840	5.1.0	Dell	x86	✔	✖	✔	✔
Nexus 10 (WiFi)	4.3.0	Samsung	armeabi-v7a	✔	✖	✖	✔
Xperia Z1 Compact	4.3.0	Sony	armeabi-v7a	✔	✔	✔	✔
Xperia Z3	4.4.4	Sony	armeabi-v7a	✔	✖	✔	✔
Xperia Z4 Tablet	5.0.2	Sony	arm64-v8a	✔	✖	✔	✖
Fever 4G	5.1.0	Wiko	arm64-v8a	✔	✖	✔	✔
Lenny 2	5.1.0	Wiko	armeabi-v7a	✔	✷	✷	✷
Pulp 4G	5.1.1	Wiko	armeabi-v7a	✔	✷	✷	✷
Rainbow 4G	4.4.2	Wiko	armeabi-v7a	✔	✖	✔	✔
qemu-system-i386	N/A	N/A	emulator	✔	N/A	N/A	N/A
qemu-system-x86_64	N/A	N/A	emulator	✔	N/A	N/A	N/A
qemu-system-armel	N/A	N/A	emulator	✔	N/A	N/A	N/A
qemu-system-aarch64	N/A	N/A	emulator	✔	N/A	N/A	N/A
Unicorn	N/A	N/A	emulator	✔	N/A	N/A	N/A
AVD 32bit (Nexus 5)	7.1.1	N/A	emulator	✔	✔	✔	✔
AVD 64bit (Nexus 5)	7.1.1	N/A	emulator	✔	✔	✔	✔

Table 5 – Overall comparison summary of isEmu and other techniques.

Technique	Average speed (ms)	Code compatibility	Detection accuracy
isEmu	0.064	176/176 (100.0%)	176/176 (100%)
H1	16.26	162/169 (95.8%)	85/169 (50.2%)
H2	26.07	169/169 (100%)	146/169 (86.3%)
H3	8.38	169/169 (100%)	132/169 (78.1%)

existing methods based on API/Property/Filesystem artifacts shows unacceptable false positives.

Other than accuracy, we also measured code execution speed for each evaluation cases. The average detection speed of `isEmu` is `0.064 ms`. On the other hand, `H1`, `H2`, `H3` shows average detection speed of `16.26`, `26.07`, and `8.38`, respectively.

Heuristics for detecting Android emulators such as `H1`, `H2`, and `H3` could be utilized more efficiently by massively combining similar artifacts and using a statistical approach. Indeed, Morpheus suggests that there are more or less 10,000 similar artifacts for Android emulator detection. While such an approach is another insightful way for anti-emulation, a large aggregation of artifacts could be time-consuming and can be relatively unreliable in other aspects such as compatibility, power consumption. In addition, `isEmu` do not require any Android permissions while other heuristics require three permissions (`READ_PHONE_STATE, ACCESS_NETWORK, INTERNET`). Table 5 summarizes overall evaluation result of `isEmu` compared to other heuristic methods.

5. Discussion

5.1. *Other methods for dynamic analysis*

In this paper, we made discussion based on assumption that attackers are equipped with emulators to analyze commercial software. However, there are other methods for analyzing an unknown binary. For example, DroidScope (Yan and Yin), TaintDroid (Enck et al., 2014), DroidBox (Lantz et al., 2012), and ANANAS (Eder et al., 2013) are Android analysis framework based on Dalvik VM manipulation rather than emulation. Analysis framework based on Dalvik VM manipulation targets Java code for their analysis and do not handle JNI codes. To trace the Java codes, such frameworks typically inserts hooks in Java bytecode level. For example, TaintDroid changes the Dalvik VM to effectively trace the behavior of an unknown Android application. Other frameworks also change the kernel to trace the application. For example, Andrubis modifies the kernel to trace the unknown application. Such approaches for analyzing the application is an effective way to gather execution traces regarding API calls, system calls. Detecting such analysis attempt could be done in many ways (we do not discuss this issue in-depth in this paper), such as examining the hash of Dalvik VM code and comparing it with a known value to figure out if the contents are changed from publicly known version.

5.2. *Cycle-level emulator*

Although there are dynamic analysis frameworks based on emulators, not all the emulators are meant for security analysis. For example, there are cycle-level simulators (Franke, 2008) which its primary goal is to debug the newly-developed SoC hardware in a cycle-accurate manner. Using such simulator, hardware-specific behavior such as *alignment-fault* can be emulated exactly same as real hardware. However, using such simulator for dynamic software analysis is inadequate because their execution speed is orders of magnitude slower than a real machine. For example, booting up a simplest embedded Linux system with such simulator takes up almost a whole day.

5.3. *Intel based android devices*

While testing the deployability of our technique (unaligned vectorization based emulator detection), we have encountered some peculiar cases where Android devices were built on top of Intel architecture. We found that Android devices such as `ASUS Memo Pad 7`, `Samsung Galaxy Tab 3`, `Dell Venue 8 7840`, and `Toshiba Excite Go` is based on Intel x86 architecture. Surprisingly, the native code of JNI which is based on ARMv7 instructions successfully operates under such configuration. Such devices use x86 based ARM emulation engine (`libhoudini`, lib) to run the ARM binary with Intel chip. Therefore, reporting such devices as emulator is technically correct, but could be considered wrong in terms of providing service to benign users. We plan to use other measures for distinguishing such peculiar cases.

5.4. *DBI*

Dynamic Binary Instrumentation (DBI) techniques such as Intel PIN tools (pin; Nethercote and Seward, 2007) are also used for analyzing and improving the runtime performance or enforcing a security policy. For supporting this, It allows a programmer to instrument code instructions at runtime. More specifically, it gains control over all control transfer instructions and thus is able to intercept an instruction and create a new instruction sequence for instrumentation. In this process, to improve efficiency, It creates the code cache to reuse instrumented instruction sequences, which makes it unnecessary to generate the instruction sequences again. As QEMUs TCG translates and generates a translated instruction sequence in basic block granularity, DBI frameworks also generate a code cache as a basic block unit.

It seems like that our method that utilizes the unaligned access mechanism can be applied to DBI frameworks. However, there is the difference between them. Unlike emulators

benign users from the analysis engine, we plan to detect such peculiar cases in another way.

that translate an instruction set into host instruction set, DBI frameworks directly copy the executable code and run it natively with minimal change. In other words, an emulator is able to execute a binary which has a different ISA from the host machine. (e.g., running arm binary on x86 machine) However, DBI frameworks only execute a binary which has the same ISA as the host machine. Therefore, a translated misaligned vectorization instruction behaves just like running on the real machine, not like an emulated environment. However, instead of the way to detect DBI using unaligned access mechanism, the other methods (e.g., race condition, TB cache thrashing) can be utilized for DBI detection. It is achieved by the DBI code cache technique that translates and generates the code cache in a basic block unit.

5.5. *Anti-anti VM*

Regarding bypassing the detection, anti-anti-vm technique (deactivating the detection mechanism) can be considered. In general, anti-anti-vm tools additionally implements API wrappers (ant), or modifies instruction semantics (ant) to manipulate the information that the anti-vm detection technique expects. At this point, no publicly known anti-anti-vm detection tools considered our anti-emulation technique. However, in theory, any anti-emulation attempts can be prevented the same as any obfuscated binary can be eventually analyzed. The problem is the cost of this prevention/analysis. In the case of isEmu, the emulator can prevent the detection attempt if the alignment-fault feature is accurately implemented in the emulator code translation process. Implementing such redundant features regarding alignment-fault requires re-defining the translation codes of various memory access instructions. In the case of the context-switch based-detection, the anti-anti-vm technique can be implemented by making emulator update its program counter after executing each instruction, which reforms QEMUs TCG framework and deprives the benefit of basic-block-level translation. Overall, the complete emulation of all hardware architectures may lead to huge performance degradation and also requires many engineering efforts.

6. Related work

To date, various anti-emulation techniques have been proposed to detect the existence of dynamic analysis systems in emulated environments (Raffetseder et al., 2007; Yokoyama et al., 2016; Thompson et al.; sym, 2006; Dinaburg et al., 2008; Garfinkel et al., 2007; Jing et al., 2014; Lau and Svajcer, 2010; Omella, 2006; Pék et al., 2011; Petsas et al., 2014; Vidas and Christin, 2014; Wang et al., 2012). In general, anti-emulation techniques exploit the discrepancies between real devices and emulators. As Petsas et al. suggested, these techniques can be categorized into three classes: static, dynamic, and intricacy-based. In general, static detection techniques use filesystem artifacts, hard-coded strings. Dynamic detection technique gathers runtime information such as sensor data value. Finally, intricacy-based techniques exploits the architectural discrepancy such as instruction behavior. isEmu can be categorized as intricacy-based technique.

6.1. *Morpheus*

Static detection approach fingerprints artifacts (e.g., APIs, files, etc.) that are initialized with fixed value while system initialization. For example, one approach can retrieve current build information using android.os.Build class which has HARDWARE and PRODUCT field. If an application runs on a naive Android emulator, this HARDWARE and PRODUCT field has goldfish and google_sdk, respectively. These static-based techniques can detect the emulator quickly, but they may be easily bypassed or unreliable as our evaluation demonstrated. However, due to a large number of artifacts, such detection attempts are still dominant as shown in Morpheus (Jing et al., 2014). Jing et al. proposed Morpheus framework that systematically and automatically collects visible artifacts and ranks detection heuristics in order of effectiveness. More specifically, it retrieves visible artifacts and their contents from real devices and emulators. In the analysis phase, Morpheus classify the artifacts and its contents into Type E and Type D heuristics, which indicates emulators and real devices, respectively. Overall, it uncovers more than 10,000 detection heuristics and ranks top 10 detection heuristics in each File, API and Property type.

6.2. *SandPrint*

SandPrint (Yokoyama et al., 2016) discussed various artifacts for disclosing the application running environment. Five static-detection primitives (installation, network, etc.) are used for gathering various artifacts. Afterward, SandPrint conducts a comprehensive analysis regarding the detection feature selection based on clustering and learning methods. While SandPrint focuses on revealing sandboxed running environment, we focus to detect emulators based on binary-translation. Another difference between this work and ours is that SandPrint is based on the Windows environment and its main discussion is predicated under the assumption that sandboxes are designed for malware. On the other hand, isEmu assumes Android environment, and it is designed under the assumption that malicious users cracking the intellectual property of software via code-level emulator.

6.3. *Detecting System Emulator*

Detecting System Emulator (Raffetseder et al., 2007) demonstrated that emulators are not necessarily stealthier than virtual machines, as suggested by Bayer et al. (2006) and Vasudevan and Yerraballi (2006). They demonstrated various kernel-level technique that measures the relative timing performance of privileged instructions. In particular, the paper shows that in a real machine environment, the CR3 register access time is similar to the access times of other registers; however, in the QEMU environment, the CR3 access time is orders of magnitude slower than that of the other registers. On the other hand, the cache invalidation speed in a real machine environment is orders of magnitude slower than that of the emulator. The paper additionally discussed other discrepancies of CPU behavior regarding instruction bugs, model specific registers, and alignment checking in x86. While the previous discussion regarding alignment check based detection requires kernel modification thus unfit for commercially deployed application, unaligned vectorization based detection in this is paper is solely based on user-privileged instructions without any kernel dependency therefore suited for commercial application deployment.

6.4. *Virtualization detection: new strategies and their effectiveness*

Thompson et al. suggested the *Counter-Based Timing* mechanism to distinguish QEMU. This method demonstrated that QEMU shows abnormally faster timing performance of the CPUID instruction compared to nop. Their experiment showed that the timing performance of the CPUID instruction is approximately 10 times slower than nop in the QEMU environment; however, the ratio of CPUID to nop reached 200 from a native environment. This detection technique can be rather trivially mitigated by slowing down the execution speed of the CPUID instruction by using QEMU thus inappropriate for commercially used software.

6.5. *Attacks on more virtual machine emulators*

Symantec (sym) published a paper (sym, 2006) that broadly covers the execution environment of fingerprinting techniques. The paper surveys several virtual machines including VMWare (vmw, 2006), Xen (xen), Parallels (par), VirtualBox (vir), Bochs (boc), and QEMU (Bellard., 2005). This paper mainly focused on the behavioral difference caused by implementation bugs in specific instructions. Their work indicates that there are a number of discrepancies in virtualized and emulated environments running on real hardware. However, emulator detection based on such discrepancies are less accurate to be used for commercial products.

6.6. *Compatibility is not transparency: VMM Detection Myths and Realities*

Other than the timing attack, various techniques have been proposed to distinguish the running environment for virtual and native platforms. *VMM Detection Myths and Realities* (Garfinkel et al., 2007) demonstrated logical, resource and timing discrepancies to reveal the virtual machine environment. The logical and resource discrepancies exploit the difference between a hardware interface and shared physical resources. The timing discrepancy measures the timing difference of instructions between the virtualized and native environments. The indicators for timing discrepancy included increased clock cycles by using device virtualization, additional page faults induced by memory virtualization, and privileged instructions overhead by using CPU virtualization.

6.7. *Defeating the transparency features of dynamic binary instrumentation*

Previously, there are some researches for DBI detection (Li and Li, 2014). DBI frameworks such as Dynamorio should provide a transparent environment for a binary as they do not translate the entire set of instruction (mostly branches). However, since the DBI framework shares the virtual address space with the instrumented binary. Therefore, there can be discrepancies between a native execution and instrumented one. For example, DBI frameworks can change the program resources regarding signal handlers, memory, file descriptors, and loaded libraries, etc. Such change can be heuristically detected in many ways. Identifying such instrumentation is conceptually similar to anti-emulation, however, in this paper, we mainly focus on full ARM system emulation for mobile applications.

6.8. *Ether vs nEther*

Ether (Dinaburg et al., 2008) demonstrated the mitigation of a timing attack by relying on the local system timer. The mitigation was based on a method that feeds the malware with the adjusted time information. When a request for system time occurred, Ether adjusted the time information to exclude the amount of time spent, whereas the virtualization/emulation related components such as the exception handler routine. Ether also mentioned that their mitigation approach fails if the malware succeeds in retrieving time information from the system external clock such as the network time. However, after Ether was published, nEther (Pék et al., 2011) showed that the timing attack that was supposedly prevented by Ether can still be disclosed by measuring other timing sources such as CPU RPM, and DMA.

7. Conclusion

In this paper, we focus on finding an anti-emulation technique for protecting the intellectual property of corporate software products in large-scale deployment. In particular, we mainly discussed three detection techniques leveraging architectural internals of hardware and emulator design (context switch, translation cache, and misaligned vectorization). The presented techniques outperformed previously known methods in many terms. We have conducted various experiments regarding the proposed anti-emulation techniques to find a method suited for our purpose. In the end, we conclude that *misaligned vectorization* based detection is a promising anti-emulation technique suited for protecting commercial software and developed isEmu. To verify the practicality and deployability of our technology, we tested isEmu and three existing heuristic detection methods against 176 various models of mobile devices and several emulators. isEmu is a universal Android emulator detection app that works without any Android permissions. The evaluation result so far suggests that misaligned vectorization based anti-emulation guarantees the correctness (peculiarities are discussed in Section 5) with better performance. We are currently porting this technique for Samsung's next-generation mobile security feature.

Acknowledgement

This research was supported by Samsung Research (Electronics), National Research Foundation of Korea (NRF) and

Human Resource Development Project for Brain Scouting, IITP (Institute for information & communications Technology Promotion), ICT & Future Planning, and Office of Naval Research (NRF-2017R1A2B3006360, IITP-2017-0-01889, IITP-2017-0-01853, N00014-18-1-2661).

Appendix A. Proof-of-concept source codes

```
// gcc -o race race.c -lpthread -m32
#include <stdio.h>
#include <pthread.h>

#define NOP4 "nop\n""nop\n""nop\n""nop\n"
#define NOP16 NOP4 NOP4 NOP4 NOP4
#define NOP64 NOP16 NOP16 NOP16 NOP16
#define NOP256 NOP64 NOP64 NOP64 NOP64
#define NOP1024 NOP256 NOP256 NOP256 NOP256

unsigned int lock = 0;
void* race(void* p){
    while(1){
        asm(
                "mov␣%0,␣%%eax\n"
                "inc␣%%eax\n"
                "mov␣%%eax,␣%1\n"
                :"=r"(lock) :"r"(lock)
           );
        asm(
                NOP1024
           );
        asm(
                "mov␣%0,␣%%eax\n"
                "dec␣%%eax\n"
                "mov␣%%eax,␣%1\n"
                :"=r"(lock) :"r"(lock)
           );
    }
    return 0;
}

int main(){
    FILE *out;
    int i = 0;
    pthread_t p;
    pthread_create(&p, 0, race, 0);
    pthread_create(&p, 0, race, 0);
    while( lock < 2 ){
        printf("This␣is␣QEMU\r");
    }
    printf("This␣is␣not␣QEMU!\n");

    return 0;
}
```

Listing 1 – Context switch based detection.

```
// gcc -o self self.c -m32
#include <stdio.h>
#include <sys/mman.h>

unsigned long long rdtsc(){
        asm("rdtsc");
}

void modify1(){
        asm("movl␣$0xc031c031,␣0x66666000");
        asm("movl␣$0xc3c3c3c3,␣0x66666004");
}

void modify2(){
        asm("movl␣$0x90909090,␣0x66666000");
        asm("movl␣$0x909090c3,␣0x66666004");
}

void self(){
        if((*((char*)0x66666000)) == 0x90){
                modify1();
        }
        else{
                modify2();
        }
        asm("push␣%eax");
        ((void (*)(void))0x66666000)();
        asm("pop␣%eax");
}

void dummy(){
        int a, b;
        for(a=0; a<100; a++){
                b += a;
        }
}

void fix(){
        asm("push␣%eax");
        dummy();
        asm("pop␣%eax");
}

unsigned int anchor=0;
void run(const char* inst_name, void (*f)(), int l){
        register int i;
        unsigned long long t2, t1;
        i = l;
        t1 = rdtsc();
        while(1){
                f();
                if(i==0) break;
                i--;
        }
        t2 = rdtsc();
        printf("%s␣:␣%llu\n", inst_name, t2-t1);
}

int main(int argc, char* argv[]){
        void* p = mmap((void*)0x66666000, 0x1000,
            PROT_WRITE|PROT_READ|PROT_EXEC, MAP_ANONYMOUS|
            MAP_SHARED, -1, 0);
        printf("allocated␣at␣:␣%p\n", p);
        run("self", self, 1000);
        run("fix␣", fix, 1000);
        run("fix␣", fix, 1000);
        run("self", self, 1000);
        return 0;
}
```

Listing 2 – TB cache based detectirrron.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <ucontext.h>

void AlignTrapHandler(int signum, siginfo_t *info, struct
      ucontext* sc){
     printf(”hardware!\n”);
     exit(0);
}

int main(void){

          // setup align trap handler
          struct sigaction act;
          memset(&act, 0, sizeof(act));
          act.sa_sigaction = AlignTrapHandler;
          act.sa_flags = SA_SIGINFO;
          sigaction(SIGSEGV, &act, NULL);

     // trigger unaligned vectorization
     __asm__(”mov␣%rsp,␣%rax\n”
          ”inc␣%rax\n”
          ”movntps␣%xmm0,(%rax)”);

     printf(”emulator!\n”);
          return 0;
}
```

Listing 3 – Unaligned access based detection.

REFERENCES

Alzaylaee MK, Yerima SY, Sezer S. Emulator vs real phone: Android malware detection using machine learning. In: Proceedings of the 3rd ACM on international workshop on security and privacy analytics. ACM; 2017. p. 65–72.

Amazon. Amazon Web Services (AWS), 2006. https://aws.amazon.com/.

Android emulator detection tool, 2017. https://github.com/CalebFenton/AndroidEmulatorDetect.

anti -anti-vm-detection-dll, 2018. https://github.com/ExpLife0011/anti-anti-vm-detection-dll-1.

Anubis. Analyzing unknown binaries, 2012. http://anubis.seclab.tuwien.ac.at.

Attacks on virtual machine emulators. In: Proceedings of AVAR conference on symantec advanced threat research, Auckland, 2006.

Bayer U, Kruegel C, Kirda E. Ttanalyze: a tool for analyzing malware. In: Proceedings of the 15th annual conference of the european institute for computer antivirus research (EICAR). FREENIX Track; 2006. p. 41–6.

Bellard F. Qemu, a fast and portable dynamic translator. In: Proceedings of USENIX annual technical conference. FREENIX Track; 2005. p. 41–6 doi:10.1109/ARES.2009.116.

Bochs : the open source IA-32 emulation project, 2001. http://bochs.sourceforge.net/.

Collberg C. The case for dynamic digital asset protection techniques. Department of Computer Science, University of Arizona; 2011. p. 1–5.

Conley J, Andros E, Chinai P, Lipkowitz E. Use of a game over: emulation and the video game industry, a white paper. Northwest J Technol Intell Prop 2003;2:261.

Defeating malwares anti-VM techniques (CPUID-based instructions). https://rayanfam.com/topics/defeating-malware-anti-vm-techniques-cpuid-based-instructions/.

Dinaburg A, Royal P, Sharif M, Lee W. Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM conference on computer and communications security, CCS; 2008. p. 51–62.

Easy to detect android emulator, 2017. https://github.com/framgia/android-emulator-detector.

Eder T, Rodler M, Vymazal D, Zeilinger M. Ananas-a framework for analyzing android applications. In: Proceedings of 2013 eighth international conference on availability, reliability and security (ARES). IEEE; 2013. p. 711–19.

Enck W, Gilbert P, Han S, Tendulkar V, Chun B-G, Cox LP, Jung J, McDaniel P, Sheth AN. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans Comput Syst 2014;32(2):5.

Franke B. Fast cycle-approximate instruction set simulation. In: Proceedings of the 11th international workshop on software & compilers for embedded systems ACM; 2008. p. 69–78.

Garfinkel T, Adams K, A W, Franklin J. Compatibility is not transparency: VMM detection myths and realities. In: Proceedings of the 11th workshop on hot topics in operating systems (HotOS-XI), 2007. doi: 10.1109/ARES.2009.116.

Jing Y, Zhao Z, Ahn G-J, Hu H. Morpheus: automatically generating heuristics to detect android emulators. In: Proceedings of the 30th annual computer security applications conference ACM; 2014. p. 216–25.

Lantz P, Desnos A, Yang K. Droidbox: Android application sandbox, 2012.

Lau B, Svajcer V. Measuring virtual machine detection in malware using dsd tracer. J Comput Virol 2010;6(3):181–95.

Lee S. A study on android emulator detection for mobile game security, 2014.

Li X, Li K. Defeating the transparency features of dynamic binary instrumentation: the detection of dynamorio through introspection, 2014. Talk at Black Hat.

Libhoudini. The default ARM translation layer for x86, 2017. https://github.com/Rprop/libhoudini.

Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of ACM sigplan notices. ACM, 42; 2007. p. 89–100.

Omella AA. Methods for virtual machine detection, Grupo S21sec Gestión SA, 2006.

Oracle VM virtualbox, 2007. https://www.virtualbox.org/.

Pék G, Bencsáth B, Buttyán L. nether: in-guest detection of out-of-the-guest malware analyzers. In: Proceedings of the fourth European workshop on system security, ACM; 2011. p. 3.

Parallels - virtualization and automation solutions for desktops, servers, hosting, SaaS, 1999. http://www.parallels.com/.

Petsas T, Voyatzis G, Athanasopoulos E, Polychronakis M, Ioannidis S. Rage against the virtual machine: hindering dynamic analysis of android malware. In: Proceedings of the seventh European workshop on system security, ACM; 2014. p. 5.

Pin. A dynamic binary instrumentation tool, 2012. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

Quynh NA, Vu DH. Unicorn – the ultimate cpu emulator, 2015. http://www.unicorn-engine.org/.

Raffetseder T, Kruegel C, Kirda E. Detecting system emulators. In: Proceedings of international conference on information security, Springer; 2007. p. 1–18.

Strazzere T. Dex education 201: anti-emulation, HITCON2013, 2013.

Symantec Corp, 1995. https://www.symantec.com/.

Thompson C, Huntley M, Link C. Virtualization detection: new strategies and their effectiveness.

VMware. Inc., 2006. http://www.vmware.com/.

Vasudevan A, Yerraballi R. Cobra: fine-grained malware analysis using stealth localized-executions. In: Proceedings of the 2006 IEEE symposium on security and privacy, IEEE; 2006. p. 15–pp.

Vidas T, Christin N. Evading android runtime analysis via sandbox detection. In: Proceedings of the 9th ACM symposium on information, computer and communications security, ACM; 2014. p. 447–58.

Wang JB, Lian YF, Chen K. Virtualization detection based on data fusion. In: Proceedings of the 2012 international conference on computer science and information processing (CSIP), IEEE; 2012. p. 393–6.

Xen T., Project, the powerful open source industry standard for virtualization, 2003. http://www.xenproject.org/.

Yan LK, Yin H. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. Usenix Security; 2012.

Yokoyama A, Ishii K, Tanabe R, Papa Y, Yoshioka K, Matsumoto T, Kasama T, Inoue D, Brengel M, Backes M, et al. Sandprint: fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In: Proceedings of international symposium on research in attacks, intrusions, and defenses, Springer; 2016. p. 165–87.

Daehee Jang received the B.S. degree in Computer Engineering from Hanyang University, South Korea, in 2012. He also received the M.S. degree in Information Security from Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2014. He is currently working toward the Ph.D. degree at the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). He is the founder of pwnable.kr wargame. His research interests include software vulnerability, operating system, and anti-emulation.

Yunjong Jeong received the B.S. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2017. He is currently working toward the M.S. degree at the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). His research interests include software vulnerability, compilers, and Intel SGX.

Seongman Lee received the B.S. degree in Computer Science from Chungnam University in 2015. He also received the M.S. in the Graduate School of Information Security at Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2017. He is currently pursing his Ph.D. at the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). His research interest includes software exploitation mitigation.

Minjoon Park received the B.S. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2016. He also received the M.S. degree in Information Security from Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2018. He is currently working toward the Ph.D. degree at the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). His research interests include software vulnerability, operating system, cryptography.

Kuenhwan Kwak received the B.S. degree from Soongsil University and M.S. in Computer Science from Korea Univerity, Seoul. His research interest includes mobile and embedded system security. He has been with Samsung Research, Seoul, Korea, where he is currently the Software Engineer of Security R&D.

Donguk Kim received the Ph.D. degree in Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea. His research interest includes application and platform Security. He has been with Samsung Research, Seoul, Korea, where he is currently the Software Engineer of Security R&D.

Brent Byunghoon Kang is currently the chief professor at the Graduate School of Information Security, and associate professor at the School of Computing at KAIST. He has also been with George Mason University as an associate professor in the Volgenau School of Engineering. Dr. Kang received his Ph.D. in Computer Science from the University of California at Berkeley, and M.S. from the University of Maryland at College Park, and B.S. from Seoul National University. Dr. Kang has been working on systems security including OS kernel integrity monitors, HW-based trusted execution environment, Code-Reuse Attack defenses, Memory address translation integrity, and Heap memory defenses.