Computers & Security 113 (2022) 102569

Contents lists available at ScienceDirect

Computers & Security

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

EmuID: Detecting presence of emulation through microarchitectural characteristic on ARM

Yeseul Choi[a], Yunjong Jeong[a], Daehee Jang[b], Brent Byunghoon Kang[a,*], Hojoon Lee[c,*]

[a] Graduate School of Information Security, KAIST, South Korea
[b] Department of Convergence Security Engineering, Sungshin Women's University, South Korea
[c] Department of Computer Science and Engineering, Sungkyunkwan University, South Korea

ARTICLE INFO

Article history:
Received 30 November 2020
Revised 26 October 2021
Accepted 29 November 2021
Available online 2 December 2021

Keywords:
Software analysis
Software emulation
Emulation detection
Microarchitectural characteristics
ARM Architecture

ABSTRACT

Software emulation is at the core of efficient automated software analysis. It allows efficient use of computing resources by running multiple instances on a single machine. Also, software emulation naturally provides a strong sandboxing that contains the analyzed target software. Software emulation techniques and principles have been implemented in dynamic binary translators (DBI) and emulators used extensively in practice. *Transparency* of emulation is one of the essential aspects of emulation engines. That is, hiding the presence of emulation from the software that is being emulated is vital in many use cases of software emulation (e.g., malware analysis). Detecting the presence of emulation through various methods and preventing such exploits have been an important topic in the field. Emulation detection is commonly used in protecting commercial software against reverse engineering or abused by malware developers who intend to sabotage their malware analysis. Many works have proposed methods for emulation detection, while others introduced mitigations. In this paper, we present EmuID that exploits a peculiar microarchitectural caveat of the ARM architecture to detect emulation. Our method is accurate, implementation-agnostic, and robust. Our evaluations show that our method detects ARM execution in well-known emulation engines on ARM (i.e., ARM-on-ARM) as well as cross-architecture ARM emulation on the x86 architecture (i.e., ARM-on-x86. Also, mitigation of our approach would require non-trivial modifications to emulation engines, unlike the heuristics-based detection methods that can be readily mitigated once the mechanisms are known.

1. Introduction

Software emulation has been an integral part of large-scale and automated software analysis. Multiple instances of an emulated execution environment can be deployed on a single machine to use the computing power efficiently. Software emulation also effectively sandboxes the running program. With full control of the emulated program's execution, software emulators can aptly contain the behavior of the emulated software within a certain boundary. The principles of software emulation have been implemented in the form of dynamic binary instrumentation (DBI) and program and system emulators.

Software emulation has been utilized in practice for a long time. *Dynamic binary instrumentation (DBI)* engines such as DynamoRio (Bruening et al., 2003) and Intel PIN Tools (Luk et al., 2005) are highly sophisticated and mature and have numerous practical use cases (D'Elia et al., 2019). These tools allow flexible code injection and transformation on the emulated software and hence widely used in software analysis. QEMU (Bellard, 2005), a userspace and full-system emulator, also builds on top of the same set of software emulation principles and techniques.

Software emulation is often deployed for dynamic analysis of software that is protected through obfuscation (e.g., packing). Since the program code and data are not visible in the static analysis, automated dynamic analysis tools are built using emulation to monitor the program's behavior (D'Elia et al., 2019). Automated analysis of malware and protected software is one such example.

Many software emulation engines seek robustness against emulation detection techniques. Many malware or protected software are equipped with a so-called anti-emulation feature that checks if it is inside an emulation environment and terminates when the check reports true. A reliable method without false-positives that detects the presence of emulation is pivotal in anti emulation. From the perspectives of the developers of emulation engines, it is important to mitigate such detection and ensure emulation transparency for many use cases of software emulation.

* Corresponding authors.
E-mail addresses: brentkang@kaist.ac.kr (B.B. Kang), hojoon.lee@skku.edu (H. Lee).

https://doi.org/10.1016/j.cose.2021.102569
0167-4048/

There are many works that proposed emulation detection techniques using features that break transparency, demonstrating the importance of the issue. Many works have presented new methods that can detect the presence of emulation (Falcón and Riva, 2012; Hron and Jermář, 2014; Jang et al., 2019; Jing et al., 2014; Kirsch et al., 2018; Li and Li, 2014; Petsas et al., 2014; Polino et al., 2017; Raffetseder et al., 2007; Sun et al., 2016). In response, researchers have proposed alternative software emulation engine designs that are immune to such detection methods (Bruening et al., 2012; Polino et al., 2017). Many existing emulation detection methods depend on heuristics; the presence of various artifacts may arise due to emulation. For instance, X. Li et al. and T. Raffets et al. presented emulation detection by catching a certain artifact such as memory usage and time overhead (Li and Li, 2014; Raffetseder et al., 2007). Catching an incomplete emulation of the actual environment such as page permission and self-modifying code was also used to detect emulation (Hron and Jermář, 2014; Jang et al., 2019; Sun et al., 2016). A number of studies explored detection methods using various artifacts or behaviors (Falcón and Riva, 2012; Jing et al., 2014; Kirsch et al., 2018; Petsas et al., 2014; Polino et al., 2017). Since these methods exploit artifacts that are specific to certain emulation engines, their methods are not generalizable to different types of emulation engines.

In this paper, we present an emulation detection method called `EmuID` that takes advantage of the microarchitectural caveat of the ARM architecture. Our novel detection method shows that maintaining emulation transparency is much more challenging than previously thought.The detection method uses a meticulously crafted code that causes a peculiar cache behavior in native execution environments. The cache behavior is, however, difficult to reproduce in emulated execution environments. Our detection method is deterministic, implementation-agnostic, and robust. Additionally, `EmuID` code has a small footprint and execution time, such that it can be embedded into any application. It is deterministic because it successfully detects the presence of emulation 100 percent of the time, as our evaluation shows, and does not require repeated measurements nor a heuristically determined threshold. We also present an evaluation that shows our detection method successfully detects all well-known software emulation engines on the ARM architecture and the cross-architecture emulation engine on the x86 architecture. Our method is robust; mitigating our detection method will require non-trivial modifications to the inherent mechanisms of the software emulation engines, to the point where the mitigation would have an impact on target emulation.

We summarize our contributions as the following:

- We introduce a robust software emulation detection method called `EmuID` that takes advantage of the cache behavior of ARM architecture.
- We provide an in-depth analysis of the detection method's mechanism and its effect on the processor cache in a native vs. emulated execution environment.
- We evaluated the accuracy of the method against well-known software emulation engines on the ARM architectures and the cross-architecture emulation engine on the x86 architecture. In addition, we confirmed that there are no false positives by testing the method on 155 ARM devices.
- We discuss possible mitigations of the attack, although we concluded that none of them were practical.

The rest of this paper proceeds as the following: we provide knowledge that can help readers better understand our paper and cover the related work in Section 2. The design and key mechanism of our detection method are explained in detail in Section 4. We present the evaluation results that show the accuracy and implementation-agnostic aspects of our method. In Section 6, we explain why modifying the software emulation engines to mitigate our method would be challenging. We will conclude in Section 7.

2. Background

In this section, we explain the background information that is necessary for understanding our work. Since our work exploits the unique cache behavior of the ARM architecture that is difficult for software emulation to mimic, it is necessary that we explain the mechanism of software emulation and ARM cache coherency in a concise manner.

2.1. Software emulation

In this paper, we use the term software emulation to refer to dynamic binary instrumentation tools such as DynamoRio (Bruening et al., 2003), Pin (Luk et al., 2005), and Valgrind (Nethercote and Seward, 2007) as well as user-level/full-system emulator like QEMU (Bellard, 2005). DBI and emulators build on the same set of principles and operate in a vastly similar manner. However, we do not include *virtualization* in our definition of the term software emulation. Today's virtualization runs guest code directly on the processor with hardware-assisted virtualization features and involves only a small amount of emulation.

The techniques and principles of the two categories of software emulation are similar, but the two are geared towards specific objectives. Pin and DynamoRio focus on user program instrumentation. On the other hand, QEMU offers user program execution emulation as well as the same architecture and cross-architecture (e.g., emulating an ARM processor on x86) but does not focus on code instrumentation. The two categories of software emulation both perform a translation process on the original code – with or without instrumentation or into different or identical architecture – and execute the process's product.

In software emulation, the target program is never executed directly on the hardware. The target program's code is fetched one basic block at a time, then translated to be placed on the *translation code cache* before execution. The translation process may insert code for analysis or transform the original code with a user-given set of transformation rules. Dynamic binary translators take full control of the target program's control flow. Only translated code cache is executed and it returns its control flow back to the emulation engine. Branch, jump or call instructions whose destinations do not already reside in the code cache are instrumented to invoke the translator. In turn, the translator would fetch the next basic block to be executed and creates an instrumented version of it inside the translation cache. The translator also makes the branch or call instructions that point to the instrumented version of the target program instead of the original target.

Fig. 1 is the abstracted version of this process. This way, the emulation software maintains full control over the target program execution.

Our detection method utilizes the fact that the granularity of a cache in the translation cache is a block. To aid the reader's understanding, we will be using the following notation to denote the original code and its counterpart in the translation cache: T: B $\rightarrow$ B_T

2.2. I-Cache and D-Cache coherency on ARM

Modern architectures such as ARM and x86 employ a split cache model on L1 caches in which there are separate caches for instructions (*I-cache*) and data (*D-cache*). Such design increases the performance since the program has distinct access patterns regarding instruction fetch and data accesses. For instance, program instructions are seldom modified, unlike data. In this L1 cache de-

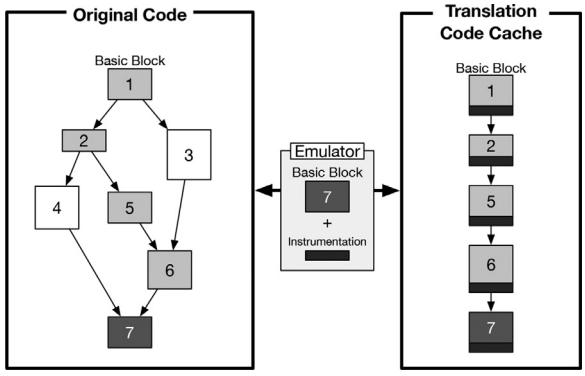


Fig. 1. Translation process in emulation. During the translation process, the original binary and the emulation code for one basic block are loaded into the translation code cache and then executed. After executing one basic block in the code cache, the emulation environment repeats the above process for the next basic block. The example figure shows that basic block 7 is converted and executed after basic blocks 1, 2, 5, and 6 executes.

sign, which is often referred to as *modified Harvard architecture* or *split cache design*, instruction fetches from memory are stored in the I-Cache and data fetches or stores are cached in the *D-Cache*.

The subject that is responsible for maintaining I-cache and D-cache coherency is architecture-dependent. In the x86 architecture, the processor itself detects incoherencies and invalidates the I-cache upon update on its counterpart in D-cache. On the other hand, all existing ARM architecture variants do not have such a feature; this means that the software is responsible for avoiding problems that can stem from the issue. Hence, programmers must maintain coherency when writing software that exhibits code modification behavior such as the ones that involve *Just-In Time (JIT)* compilation or *Self-Modifying Code (SMC)* (Jacob, 2013). I-cache invalidation can be achieved with the `SYS_cacheflush` system call on `AARCH32` and a user-level instruction such as `IC` variants of instructions are available on `AARCH64`.

3. Related work

We explain related works in the field of emulation detection. Detecting software emulation has been an interest of researchers and practitioners; malware or protected software often includes anti-emulation techniques such that it refuses to run in emulated environments to avoid being analyzed. `EmuID` presents a novel detection method that uses the unique architectural behavior of the ARM architecture. Our technique differs from the existing heuristics-based approaches that are often trivially mitigated once the heuristic is known.

3.1. Transparent software emulation

Transparency is one of the requirements for correctness and security in an emulation environment. Every emulation system has its own trade-offs between transparency and performance or emulation capabilities (Aarch64 port, 2020; Bruening et al., 2012; Hazelwood and Klauser, 2006; Nethercote and Seward, 2007; Qemu internals, 2012; Zyngie, 2015). One aspect of transparency is about the correctness of the emulation. While the emulated program is heavily modified for the sake of emulation, the apparent behavior and execution results must be identical to its native execution (Anton et al., 1998; Bruening et al., 2012). Another aspect of transparency is the robustness of emulation against emulation detectors. Since software emulation is often leveraged for automated software analysis, the robustness against detection is an important aspect of emulation engines. If it is discovered that the software is being emulated, the software that wants to hide its logic, especially malware or commercial software, can make analysis more difficult by not executing the intended behavior.

3.2. Dynamic binary instrumentation

DBI has developed steadily in response to the need to instrument and modify programs at runtime since DynInst (Buck and Hollingsworth, 2000) appeared. Pin (Luk et al., 2005), DynamoRIO (Bruening et al., 2003), and Valgrind (Nethercote and Seward, 2007) are the most well-known DBI frameworks. They are most widely used in academia and industry and support various architectures and operating systems. The Pin is a closed source framework that strongly supports the instrumentation of programs running on Intel architecture. DynamoRIO is an open-source framework that provides excellent performance and enables analysis of entire instructions and direct low-level code modification. Valgrind instruments using a generated intermediate representation that makes it portable to a variety of architectures, so there is a relative performance penalty. In addition, Frida (Karl Trygve Kalleberg, 2016), which allows users to write analysis codes in JavaScript directly, Strata (Scott et al., 2003), which provides software dynamic translation even if the architecture of the host and guest are different, and libdetox (Payer and Gross, 2011), which advances security through a design that considers transparency. In addition, various DBI frameworks (Mulliner et al., 2013; Quarkslab, 2019; Quynh, 2018) have been introduced.

3.3. Transparent software emulation

Transparency is an important the requirements for correctness and security in an emulation environment. Every emulation system has its own trade-offs between transparency and performance or emulation capabilities (Aarch64 port, 2020; Bruening et al., 2012; Hazelwood and Klauser, 2006; Nethercote and Seward, 2007; Qemu internals, 2012; Zyngie, 2015). One aspect of transparency is about the correctness of the emulation. While the emulated program is heavily modified for the sake of emulation, the apparent behavior and execution results must be identical to its native execution (Anton et al., 1998; Bruening et al., 2012).

There are several works that discuss the issue of transparency. Bruening et al. state, the further we push transparency, the more difficult it is to implement, while at the same time fewer applications require it, discusses possible solutions to each transparency problem, and suggests guidelines that should be followed in DBI design (Bruening et al., 2012). Julian et al. show that the attacker can interfere with the inspection and interposition capabilities of the emulation framework if isolation is not satisfied among the three properties (isolation, inspection, interposition) essential to the monitoring system proposed by Garfinkel et al. (2003); Kirsch et al. (2018).

In this work, we discuss mainly the emulation transparency in terms of robustness against emulation detectors. Since software emulation is often leveraged for automated software analysis, robustness against detection is an important aspect of emulation engines. Emulation detection techniques like the one we propose in this work, *break* the transparency of emulation. We propose `EmuID` in the hopes of improving the emulation transparency of future emulation engine implementations.

3.4. Detecting software emulation using heuristic features

Raffetseder et al. proposed a method to detect the system emulator using the fact that, in the emulator, the CR3 access time takes longer, and the cache invalidation speed is faster (Raffetseder et al., 2007). Daehee Jang et al. proposes a fast

and accurate anti-emulation technique that utilizes misaligning the vectorization (Jang et al., 2019). Recently, various attack techniques for detecting DBI environment for DBI detection and evasion have been published. There are several various techniques to detect DBI tools, such as DynamoRIO or Pin tool. Xiaoning Li et al. use the difference of file-related information and resource usage to detect emulation environment (Li and Li, 2014). Ke Sun et al. and Mario Polino et al. detect using the features that appear because the emulation environment uses the translation code cache (Polino et al., 2017; Sun et al., 2016). Julian Kirsch et al. detects emulation environment using code cache/instrumentation artifacts, JIT compiler over-head, and runtime environment artifacts (Kirsch et al., 2018). Francisco Falc'n et al. uses various features, including time overhead of dynamic library loading, code pointers, memory contents and permissions, and interaction with the OS (Falcón and Riva, 2012). Martin Hron et al. also mention detecting an emulation environment using a page permission violation (Hron and Jermář, 2014).

Many of the existing heuristics-based detection methods can be mitigated with a reasonable amount of effort. Once the heuristic that is used for detection is known, emulation engines can be updated in response to the detection method. However, EmuID exploits the architectural character of the ARM architecture and requires fundamental changes to emulation engines, as we will explain.

3.5. Emulation detection in android

Various anti-emulation techniques in the Android environment have also been proposed. Jingn, Yiming, et al. showed research that automatically collects and analyzes more than 10,000 heuristic artifacts for detection in the Android emulation environment (Jing et al., 2014). Petsas, Thanasis, et al. also propose anti-analysis techniques that malware can use to bypass dynamic analysis in the Android emulation environment (Petsas et al., 2014). Emulation detection technology is presented in three ways according to the analysis method in this paper: static, dynamic, and VM-related intricacies. In addition, there are two paragraphs mentioning that detection is possible using ARM's cache structure. However, the method is probabilistic, and the exact detection principle has not been investigated and tested.

3.6. Detecting virtualized environment

Thompson et al. researched to detect the virtualization environment of QEMU (Bellard, 2005), VMware (Vmware, 2020), and KVM (Kvm, 2020) using counter-based timing method (Garfinkel et al., 2007a), which uses features that have faster execution speed of specific instructions (Thompson et al., 2010). Peter Ferrie surveys attacks using differences in behavior for instruction execution on various virtual machines (Ferrie, 2007). This research analyzes known attacks on VMware (Vmware, 2020) and Virtual PC (Virtualpc, 2020) and describes new attacks and defenses against Bochs (Bochs: The open source ia-32 emulation project, 2020), QEMU (Bellard, 2005), and VirtualBox (Oracle vm virtualbox, 2020). Garfinkel et al. reveals the virtual machine environment using logical, resource, and timing discrepancies and argues that building a transparent VMM is essentially impossible (Garfinkel et al., 2007b). Pék et al. (2011) demonstrates a technique for detecting hardware-assisted virtual platforms based on CPU-specific design defects. Brengel et al. (2016) proposes a technique to detect hardware-virtualized systems using low-level timing-based mechanisms. However, techniques based on discrepancies caused by the heuristic feature suggested above are less accurate or can be bypassed by developer updates.

```
void emuID_detect() {

LAUNCHER:
    uint8_t* modified;
    int count = 0;

    //  M comes immediately after L
    modified = &launcher+LAUNCHER_LEN;

// Phase II:  Launcher L is  about  to  be  executed
LOOP:
    *modified = xor(*modified,KEY);
    count = count + 1;
    if (count != DETECTOR_LEN)
        goto LOOP;
    // Intentionally omit I-cache invalidation here
    /* invalidate_inst_cache(&DETECTOR); */

// Phase III: Launcher has finished executing.
DETECTOR:
// Phase VI: Detector Code is ready to be executed. And
     Detector Code is executed.

    // xor(0xc1035fd6,KEY) = 0xd65f03c1 (undefined)
    .inst 0xc0035fd6 // ret
    .inst 0xc0035fd6 // ret
    .inst 0xc0035fd6 // ret
    ...
}

// Returns normally in native env
// Causes a trap in emulated env
void bootstrap_emuID(){
    // EmuID_detect() code shown above
    char emuID_code[LAUNCHER_LEN + DETECTOR_LEN] = {...};

    // Load EmuID into executable memory page
    emuID = mmap(0,
                 sizeof(EmuID),
                 PROT_EXEC | PROT_WRITE | PROT_READ,
                 MAP_ANON | MAP_PRIVATE,
                 -1,
                 0);

    memcpy(emuID, emuid_code, strlen(emuid_code));
    // Phase I: Initial State. L and M are loaded into RWX
      memory.

    fptr = &emuID;
    fptr();
}
```

Fig. 2. Pseudocode of EmuID.

4. Design

In this section, we first explain the objective of EmuID and the deployment scenarios where it can play crucial role in Section 4.1. Then, we provide a multifaceted view of EmuID mechanism; We explain the key properties of our detection method in Section 4.2, present a C-like pseudocode of the implementation (Fig. 2), and an in-depth phase-by-phase illustration of EmuID's execution in Section 4.4. The phase-by-phase analysis illustrates how the contents in memory, cache, and code translation cache, in both native and emulated execution in the different stages, would appear during the execution of our detection code. Based on the in-depth analysis given in this section, we explain mitigating the attack is non-trivial in current DBI engines in Section 6.

4.1. EmuID objectives and deployment scenarios

Objectives. Our emulation detection method, called EmuID, seeks to provide a reliable and universal way to detect emulation on ARM against all well-known emulation engines that are in widespread use. Retrofitting the existing software emulation engines to nullify EmuID would be a non-trivial challenge, unlike some of the existing detections that rely on heuristics. EmuID code takes advantage of the microarchitectural characteristic of the ARM architecture. More specifically, EmuID detection code is meticu-

lously crafted to yield contrasting cache behaviors in the native and the emulated execution environment. Mitigating EmuID would require iterations of non-trivial design challenges in the software emulation engines. Even then, the additional performance overhead and the required amount of effort would render the bypassing of the detection rather difficult.

Deployment Scenarios. Emulation detection is pivotal in implementing anti-emulation features in protected software. Many protected software is packed or obfuscated to prevent static analysis. Anti-emulation actively resists dynamic analysis techniques based on software emulation by abruptly terminating the program upon detecting the presence of emulation. Hence, a reliable emulation detection technique like EmuID is a double-edged sword; it can be used to protect intellectual properties or hinder timely malware analysis.

4.2. EmuID code and properties

We implemented EmuID as a self-modifying code that terminates immediately upon detecting that it is being run in an emulated environment. The pseudocode that illustrates the behavior of EmuID is shown in Fig. 2.

The EmuID proof-of-concept is a minimal example that proves our detection method. However, it can be incorporated into a program in a stealthy manner using various techniques. This is especially true when software that employs emulation detection techniques is often highly obfuscated malware or protected commercial software. The logic of the detection code can be hidden in a benign piece of code or mutilated and obfuscated in many arbitrary ways.

Also, in many use cases of software emulation such as malware analysis, emulation engines have no option but to allow behavior that violates the W^X policy. This is because many obfuscated malware and similarly protected software use self-modifying code from hiding their behavior until runtime. For this reason, we expect that the EmuID code can easily blend in with the analyzed target software.

EmuIDcode. During the bootstrapping process, an RWX EmuID code is loaded into a memory page such that the code can modify itself (Line 37 in Fig. 2). EmuID code is composed of two components: the *launcher* (L) and the *detector* (D). L and D are positioned consecutively; D will be executed in order as L finishes. D is initially an array initialized with 0xc1035fd6 (Line 25). The value is a `ret` instruction. The launcher *modifies* or unpacks the detector by performing `xor` operations on D with a 32-bit integer KEY (Line 12). After the modification (D→ D's), the initial value 0xc0035fd6 will be unpacked into 0xd65f03c1 which is not a valid ARM instruction. The detector part of the code is designed to cause program termination in emulated execution environments.

In native execution environments, 0xc1035fd6 (`ret`) will be executed, and the detector will return without an error. On the other hand, an invalid instruction fault will be raised in emulated environments as a result of executing 0xd65f03c1. The detection code can be embedded into programs to detect and possibly deter automated emulated-based analysis. However, the content of the detection code itself does not create contrasting behavior in native vs. emulated.

EmuID code properties. EmuID detection method has unique properties that play a crucial role in making the code yield different results in native vs. emulated. First, we intentionally omit the I-cache invalidation procedure that is usually accompanied after code modification (Line 17). The ARM architecture leaves the task of synchronizing the I-cache and D-cache entries of the same cache line to software, as we explained briefly in Section 2.2. Fig. 3 illustrates the structure. Second, the length of L and D combined does not exceed the cache line size (64 bytes in the case of AARCH64). Also, L is aligned to a cache line such that L and D belong in the same cache line. This causes D to be loaded into i-cache along with L executes. Lastly, another important characteristics of the structure is that L and D are two different basic blocks, separated by a branch instruction (e.g., Line 15 in Fig. 2). The significance of these three characteristics will be further explained in this section. In all, the special properties of EmuID code can be summarized as the following:

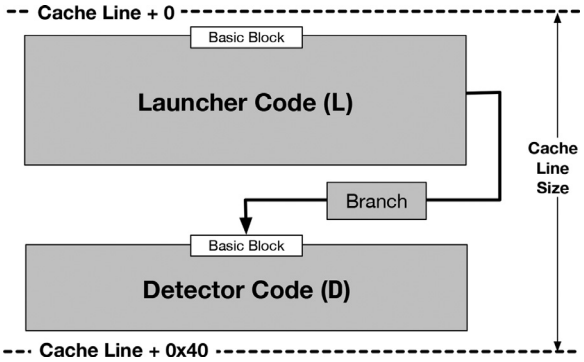


Fig. 3. EmuID Code Structure. The code is deliberately configured to have the Launcher (L) and Detector (D) that are located in different basic blocks but are loaded into the same cache line. This particular structure triggers different cache behaviors in native and emulated execution environments.

P1 I-cache invalidation is intentionally omitted after code update
P2 Launcher and Detector are placed in the same cache line
P3 Launcher and Detector are two different basic blocks separated by a branch instruction

4.3. EmuID detection mechanism

Instruction fetching in native vs. emulated. In a native execution environment, code is fetched from memory into processor I-cache with a cache line granularity. The processor then start executing the code from the I-cache. On the other hand, in the emulated environment, the software emulation adds another layer to this process. Emulation engines fetch all code before execution, makes a translated copy in the translation code cache, and executes the translated version. The emulation engine fetches the original code with basic block granularity. The particular behavior of the emulation engine may cause basic blocks in the same cache line to be separated as they are copied in the translation code cache, and EmuID takes advantage of the behavior for emulation detection.

I-cache/D-cache incoherency in ARM. Fig. 4 demonstrates how EmuID code work. When L is executed for the first time, a cache miss occurs and the cache line that L and D belongs is loaded into the I-cache (**P2**). L modifies D as it executes, and this modification is most likely reflected on the copy of D in the D-cache which is fetched from memory during the bootstrapping. This is the moment where an incoherency between the I-cache and D-cache occurs; Actually, only D in D-cache is modified to D', and D in I-cache is not modified. Hence the D remains in the I-cache. In other words, I-cache and D-cache have different codes at the same address. To resolve this incoherency, most common self-modifying codes flush the cache to reflect the modified contents to the I-cache after modification. However, we designed not to flush the cache after the D is modified to D' in order to leave D in I-cache.

After L execution is completed, D' is executed. A copy of unmodified D has been fetched as instruction into the I-cache, and D-cache contains D's, which was first loaded as data and modified by L. Since D is already located at the address of D' of I-cache, a cache hit occurs when CPU accesses D'. Therefore, D in I-cache is executed instead of D'. Due to the lack of hardware-level I-cache D-

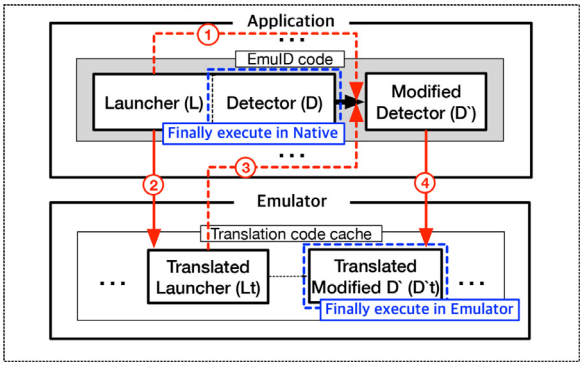


Fig. 4. EmuID overview. In the native environment, ① L modifieds D to D'. As a result, D' should be executed, but D is executed due to cache incoherency. In the emulation environment, ② L is translated to L_T in translation code cache by emulation engine. ③ L_T modifies D to D'. ④ D' is translated to D_T in translation code cache by emulation engine. As a result, D't is executed as expected.

cache syncing on the ARM architecture and property **P1** of EmuID, the processor executes D in the I-cache as a result of a cache hit. In this way, we induced D consisting of `ret` instructions to be executed unexpectedly instead of D' in the native environment. Since D is composed of a series of `ret` instructions, EmuID code essentially does nothing but simply returns the control-flow to the original caller.

Code execution and data access in emulated execution. The emulation engine intervenes with all code fetches, but not data accesses; The instructions that access data may be instrumented for various purposes, the access to data is not obstructed and reaches the in-memory original copy. This creates complexities when emulating an self-modifying code. The emulation engine must be on watch (e.g., by catching all memory modifications to the RWX pages) for modifications to code pages so that it can invalidate its code caches when the original copies of the cache are modified. We explain how this characteristic of emulation engines further complicates EmuID mitigation in Section 6.

EmuID execution in emulation environment. The process of instruction fetching in the emulation environment is different from that in the native environment. In the emulation environment, the original binary is not directly fetched into I-cache. The code block of the binary is dynamically translated with basic block granularity and fetched into the translation code cache.

After one basic block is translated and executed, the emulation environment checks the position of the next code block, and the next basic block is processed sequentially.

In our detection method, we intentionally place the L and the D in the different basic block. In order to execute the L, the emulation engine first translates the basic block of the L to the L_T and fetches it to the translation code cache. After that, the L_T in the translation code cache is fetched into the I-cache for execution. The L_T fetched in I-cache is executed by CPU. The L_T modifies the D to the D'. The L_T modifies the code of the original binary when the modification occurs without modifying the translation code cache.

After the execution of L_T is finished, the emulation engine checks whether the next basic block to be executed is in the translation code cache. Since the D', which is the next basic block to be executed, has never been fetched into the translation code cache, the emulation engine translates the D' to D't and fetches it to the translation code cache. The D't in the translation code cache is fetched into I-cache for execution and then executed by CPU. In this way, we induced D't consisting of undefined instructions that generate an error to be executed in the emulation environment, in contrast to the native environment that executes the D.

4.4. In-Depth analysis of *EmuID* execution in native vs emulated

In this section, we describe the overall detection flow and the changes in memory and cache contents per step. The location of each phase is indicated in Fig. 2.

I Launcher (L) and Detector (D) has been loaded into memory
II Launcher L is about to be executed. I-cache is loaded with the cache line that contains L
III Launcher L has finished executing
IV Detector D, which has been modified by L (D → D's), is about to be executed.

The execution flow of the detection algorithm in the native environment and the emulation environment is depicted in Figs. 5–8. Figures consist of split cache in CPU and memory in which the program running EmuID is loaded. Also, the program has L basic block and D basic block. The emulation execution in these figures shows the operation in both ARM and x86 architectures. Both ARM and x86 architectures mostly have L1 cache composed of split I-cache and D-cache, and the emulation engine translates with basic block granularity in both architectures. ARM and x86 architectures differ when it comes to cache coherency policies (manual vs. automatic). However, in the emulation environment, there is no execution that requires cache coherency, so the execution flow of EmuID in both architectures is similar.

Phase 1: Initial state. Fig. 5 shows the initial state in the native environment and the emulation environment. EmuID code has been loaded into memory and ready to be executed. The L and the D have been copied into the executable page allocated through the `mmap()` function. As a result of the copying, L and D have been fetched into the D-Cache. Up to this phase, the contents of EmuID code in memory and caches are the same in native and emulation execution.

Phase 2: L is about to execute. Fig. 6 captures the moment when L is about to execute on the processor. In native execution, an instruction fetch on L has occurred and the cache line that contains L has been placed in the I-cache. Because of P2, L and D are positioned in the same cache line, and D is loaded into the I-cache along with L at this moment.

In emulated execution, however, L is not directly executed. The emulation engine catches a *translation code cache miss* since L is executing for the first time. In turn, the engine fetches the code as data, performs necessary translations, then places it in the code cache. It should be noted that this code fetching is done *basic block by basic block*, unlike the instruction fetch performed by processors, which has a granularity of cache lines.

As a result, L_T (counterpart of L in the translation code cache) is fetched by the processor to be executed. Additionally, a subtle, but rather significant difference occurs here. Unlike the case of the native execution, D is not loaded in the I-cache along with L. This is because L and D are two different basic blocks (P3). These differences affect the consequent executions to cause completely different results in the native execution and emulated execution.

Phase 3: L has finished executing.

Fig. 7 shows when the L has finished executing. As shown in Line 12 in Fig. 2, L modifies D and the now modified D is denoted as D'. In the native execution, writes to D may be reflected on the corresponding cache line in the D-cache. This is where incoherency between the entries for D in I-cache and D-cache arise. The D-cache has received the modification, but not the I-cache. As we discussed in Section 2.2, this discrepancy is not automatically reconciled by the ARM microarchitecture. Therefore, a stale copy of D is left behind in the I-cache.

In the emulation environment, the L_T has been executed. Consequently, the D has been modified to D'. In emulated execution, data read and writes behavior is not altered unless given a specific

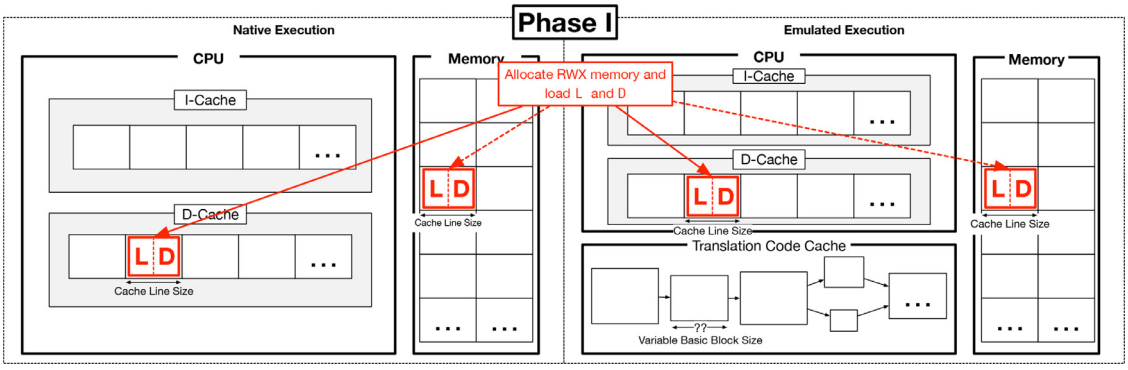


Fig. 5. Phase 1. Initial State. L and D are loaded into RWX memory. To this end, L and D are first loaded in D-cache and then updated in memory.

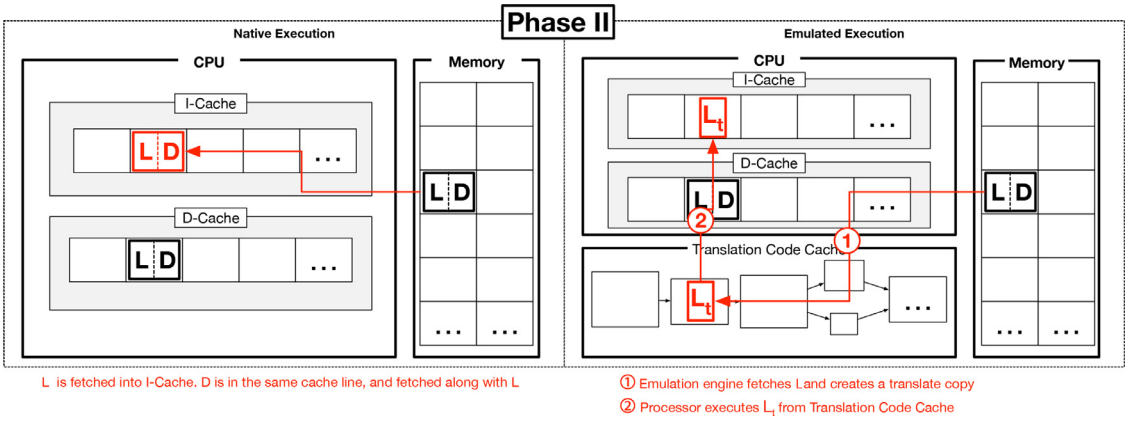


Fig. 6. Phase 2. Launcher is ready to be executed. In A, L is loaded into I-Cache. Since L and D are in the same cache line, D is loaded along with L as a side effect. In B, L is fetched, translated to become L_T, which is then placed in the code cache. L_T is loaded into I-cache (Notice that D is not loaded into I-cache).

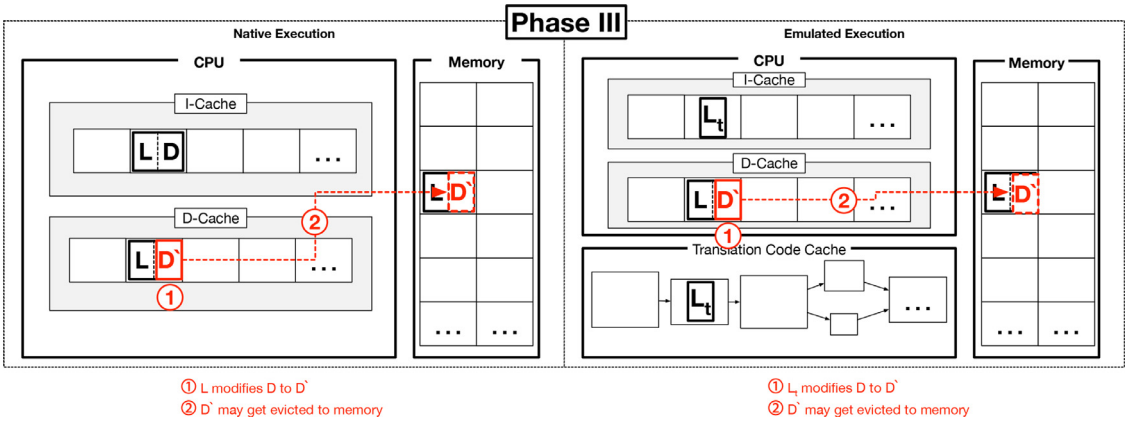


Fig. 7. Phase 3. Launcher has finished executing. The D is modified to the D'. In A, since there has been no explicit I-cache invalidation, D in I-cache is stale from this point on.

translation rule. Hence, the modifications to D performed by L are not different from those of the native execution. However, the difference from the native execution that must be pointed out here is that D has not been loaded into the I-cache. Therefore, when D is finally executed by the processor, it will cause an I-cache miss, as we will explain in the next phase.

Phase 4: D' is about to execute.

In this phase, the `EmuID` code manifests its contrasting behaviors in native vs. emulated execution environments. In native execution, a cache hit occurs on a copy of D in the I-cache; the memory writes to D has only been reflected on the copies in the D-cache or memory, but a I-cache hit occurs on the stale copy in the I-cache as intended with P1. The final result of the `EmuID` code execution in native execution is essentially an empty function; D, rather than D', executes and a `ret` instruction in D immediately hands over the control-flow back to the caller.

Meanwhile, in the emulated execution, `D't` is executed. P3 has separated L and D such that only L has been fetched through the instruction fetch sequence of the emulation engine to leave L_T in translation code cache and the I-cache. D, on the contrary, has never been fetched as an instruction. Hence when D is executed as L terminates, it is fetched for the first time to be first translated and executed. This means that the emulation engine will read D' (which has been already modified), from either D-cache or memory to create `D't`. `D't` is executed as the final result, and the un-

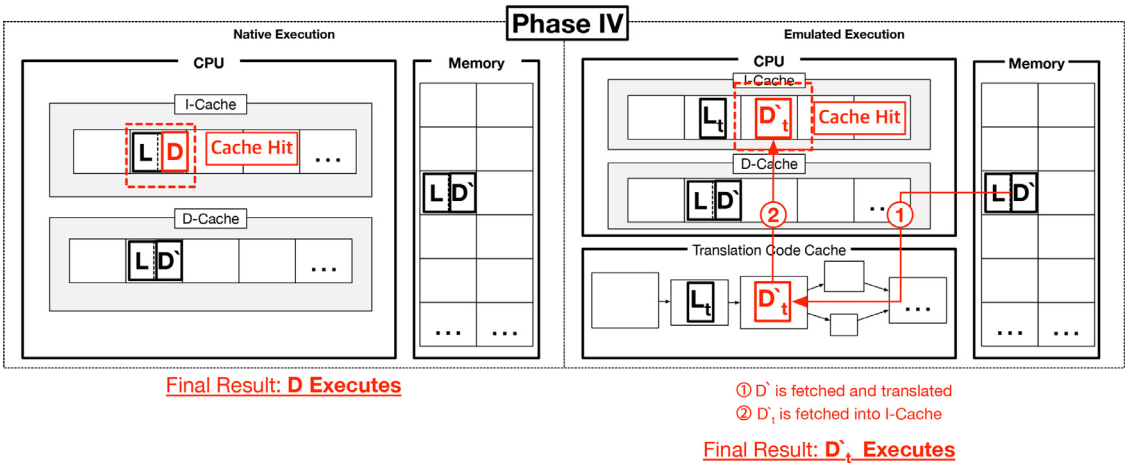


Fig. 8. Phase 4. D' is ready to be executed. In A, the processor tries to execute D', finds its address in I-cache (cache hit). Not knowing that D in I-cache is stale, the processor executes it. In B, the emulation engine has never executed D, and it is not found in the translation code cache (TCC miss). The code cache for D must be created. Thus, the resulting code cache in the translation code cache would be `D't`. And then, `D't` is executed on the processor.

defined instruction in D', which essentially works as a `trap`, terminates the execution.

5. Implementation and evaluation

We conducted a thorough evaluation to prove the effectiveness and accuracy of EmuID. First, we tested EmuID against three emulation engines on four different ARM-based devices. The detection of a cross-architect emulation engine on two different x86-based devices was also tested.

This experiment is intended to show that the detection method of EmuID is generalizable to different emulation engine implementations It also indicates that the EmuID is applicable to ARM emulation engine running on other architecture.

The experiment should also test the occurrence of possible *false-negatives* by EmuID detection. Second, we also ran the EmuID detection test on 159 types of ARM-based devices to measure any occurrences of *false-positives*. Additionally, we measured the time consumed for each detection.

5.1. EmuID implementation

Fig. 9 is EmuID code written for the 64-bit ARM and x86 architecture we used for the evaluation, whose operation is detailed in Section 4. The EmuID code is implemented separately for both 32-bit and 64-bit architecture due to ISA and cache line size differences. However, the effectiveness and accuracy of the two implementations are identical, as assured by our experiment results. The 32-bit implementation of EmuID can be found in Appendix A. EmuID has a small footprint so that it can be embedded in other programs to detect emulation. A developer can bootstrap EmuID by copying it into an RWX page and executing it, as shown in section Section 4.2. As a final execution result, EmuID simply returns to its original caller in native execution, but it will cause an *illegal instruction* fault in emulated execution.

From line 7 to line 20 is the launcher (L), and from line 27 is the detector (D). In L, The register `x0` is used as a position of the D. The register `x1 (w1)` is used as a temp register for `xor`. The register `x2 (w2)` is used as a key using for `xor`. The register `x3` is used as a loop counter. The value of the key for `xor` is set 1 in line 9, and the loop counter for `xor` is set in line 10. The position of the D is set in lines 11–12.

After the register values are set, the L modifies the D, which is initially a series of `ret` instructions. The modified D' is now a

```
1
2  // x0 : Pointer to Code Position
3  // x1 : Temp Register for XOR
4  // x2 : XOR Key
5  // x3 : Loop Counter
6
7  0x00000000:  00 04 00 91    add  x0, x0, #1
8  0x00000004:  42 00 02 CB    sub  x2, x2, x2
9  0x00000008:  42 04 00 91    add  x2, x2, #1
10 0x0000000c:  83 02 80 D2    movz x3, #0x14
11 0x00000010:  00 00 00 10    adr  x0, #0x10
12 0x00000014:  00 A0 00 91    add  x0, x0, #0x28
13 0x00000018:  01 00 40 B9    ldr  w1, [x0]
14 0x0000001c:  21 00 02 4A    eor  w1, w1, w2
15 0x00000020:  01 00 00 B9    str  w1, [x0]
16 0x00000024:  00 10 00 91    add  x0, x0, #4
17 0x00000028:  63 04 00 D1    sub  x3, x3, #1
18 0x0000002c:  7F 00 00 F1    cmp  x3, #0
19 0x00000030:  41 FF FF 54    b.ne #0x18
20 0x00000034:  1F 20 03 D5    nop
21
22 // XOR every word (4byte) with 0x1 beyond this point
23 // Convert ret instructions to undefined instructions
24 // xor(0xc1035fd6,0x1) = 0xd65f03c1 (undefined instruction
       )
25
26 // Detector (D)
27 0x00000038:  C0 03 5F D6   ret
28 0x0000003c:  C0 03 5F D6   ret
29 0x00000040:  C0 03 5F D6   ret
30 0x00000044:  C0 03 5F D6   ret
31 ...
```

Fig. 9. Self-Modifying code of EmuID for the 64-bit ARM and x86 architecture. The code before line 20 is the L that sets the register and dynamically modifies the D to the D'. From line 27 is the detector (D). The branch instruction in line 20 was used to make the L and the D belong to different basic blocks. The value of the key for XOR is set in line 9 and the loop counter for XOR is set in line 10. The position of the D is also set in lines 11–12. Lines 13–19 contain the contents of the loop to XOR with the `ret` instruction and the key.

series of (`0xd65f03c1`). Lines 13–19 contain the contents of the loop for `xor`. In this loop, the `ret` instructions of D will be unpacked through `xor` operation with the key. Each `ret` instruction is modified to the undefined instruction (`0xd65f03c1`) repeatedly. When L finishes executing, now D' will start executing. In the case of native execution, a stale copy of D in i-cache will execute, and the EmuID code will return immediately. On the other hand, in emulation execution, D' (or more precisely `D't`) will execute and get an *illegal instruction* fault.

In short, we implemented the detection code that causes different execution results in the ARM native environment and emulation environment with just one self-modifying code. This code is

Table 1
False-negative test results from 1000 trials of EmuID detection on emulation engines.

Emulation Engine	Device	Processor Arch	Core Name	Cache Line	OS	False-negative Rate
DynamoRIO	RPI 2 Model B v1.1	armeabi-v7a	Cortex-A7	32 Bytes	Ubuntu MATE 18.04.2	0%
DynamoRIO	RPI 3 Model B v1.2	arm64-v8a	Cortex-A53	64 Bytes	Ubuntu MATE 18.04.2	0%
DynamoRIO	Hikey 960	arm64-v8a	Cortex A73, Cortex A53	64 Bytes	Leuuntu 16.04	0%
DynamoRIO	Juno r0	arm64-v8a	Cortex-A57, Cortex-A53	64 Bytes	Openembedded	0%
Valgrind	RPI 2 Model B v1.1	armeabi-v7a	Cortex-A7	32 Bytes	Ubuntu MATE 18.04.2	0%
Valgrind	RPI 3 Model B v1.2	arm64-v8a	Cortex-A53	64 Bytes	Ubuntu MATE 18.04.2	0%
Valgrind	Hikey 960	arm64-v8a	Cortex A73, Cortex A53	64 Bytes	Lubuntu 16.04	0%
Valgrind	Juno r0	arm64-v8a	Cortex-A57, Cortex-A53	64 Bytes	Openembedded	0%
QEMU-user	RPI 2 Model B v1.1	armeabi-v7a	Cortex-A7	32 Bytes	Ubuntu MATE 18.04.2	0%
QEMU-user	RPI 3 Model B v1.2	arm64-v8a	Cortex-A53	64 Bytes	Ubuntu MATE 18.04.2	0%
QEMU-user	Hikey 960	arm64-v8a	Cortex A73, Cortex A53	64 Bytes	Lebuntu 16.04	0%
QEMU-user	Juno r0	arm64-v8a	Cortex-A57, Cortex-A53	64 Bytes	Openembedded	0%
QEMU-system-arm	RPI 2 Model B v1.1	armeabi-v7a	Cortex-A7	32 Bytes	Ubuntu MATE 18.04.2	0%
QEMU-system-arm	RPI 3 Model B v1.2	arm64-v8a	Cortex-A53	64 Bytes	Ubuntu MATE 18.04.2	0%
QEMU-system-arm	Hikey 960	arm64-v8a	Cortex A73, Cortex A53	64 Bytes	Lebuntu 16.04	0%
QEMU-system-arm	Juno r0	arm64-v8a	Cortex-A57, Cortex-A53	64 Bytes	Openembedded	0%
QEMU-user	Intel Desktop	Intel Coffee Lake	Core i9-9900K	64 Bytes	Ubuntu 18.04.5	0%
QEMU-user	AMD Desktop	AMD Zen 2	Ryzen Threadripper 3990X	64 Bytes	Ubuntu 20.04.2	0%
QEMU-system-arm	Intel Desktop	Intel Coffee Lake	Core i9-9900K	64 Bytes	Ubuntu 18.04.5	0%
QEMU-system-arm	AMD Desktop	AMD Zen 2	Ryzen Threadripper 3990X	64 Bytes	Ubuntu 20.04.2	0%

intended to serve as a proof-of-concept. Fig. 9 was executed in the application as a shellcode. Fig. 9 is applicable when the cache line size is 64 bytes, but in the case of 32 bytes, the shellcode is larger than the cache line size and is not loaded in one cache line. For this case, if the cache line size is 32 bytes, the thumb mode assembly code of self-modifying code, which performs a similar operation as the code in Fig. 9, is executed. In the end, our technique was applicable to both 32 bytes and 64 bytes without being limited by cache line size.

5.2. Evaluation of emulation detection accuracy

We evaluated the generalizability and accuracy of the EmuID detection method in terms of false-negatives on three widely used emulation engines (DynamoRio, Valgrind, and QEMU (QEMU-user and QEMU-system-arm)) on four ARM-based devices with varying architectures. Especially for QEMU, which provides cross-architecture emulation, we additionally tested it on two different x86-based devices.

Table 1 illustrates the result of the experiment. We ran 1000 trials for each emulation engine and device combinations. As our results illustrate, EmuID was able to detect all tested emulation engines across four devices without any false-negatives. In addition, EmuID was able to detect emulation engines for ARM running in x86 architectures without false-negative.

This experiment proves four aspects of EmuID effectiveness. First, EmuID is 100% accurate in emulation detection. If an emulation detection scheme has non-zero false-negative, one can simply run emulation indefinitely until it fails. Hence, this is a required virtue for all emulation detection schemes, and EmuID is proved to satisfy the requirement. However, a 100% accurate detection rate does not necessarily mean that the scheme is robust. That is, if a quick fix to the emulation engine can nullify the mechanism by which the detection scheme detects the emulation, the scheme is not robust. We discuss why mitigating EmuID is a daunting challenge for emulation engines in Section 6. Second, EmuID is implementation-agnostic. EmuID successfully detected all three emulation engines we used in the experiment. This is because EmuID exploits the discrepancy between the caveat of the ARM microarchitecture and the underlying shared principle in software emulation implementations. Third, EmuID successfully detects emulations on both 32-bit and 64-bit devices. There are differences between 32-bit and 64-bit, such as the cache line size (32 bytes vs. 64 bytes) and the instruction set architecture (the emulation engines are compiled to 64-bit executables for 64-bit devices). However, such factors did not affect the detection accuracy of EmuID. Fourth, EmuID is able to detect a cross-architecture emulation engine. Since I-cache and D-cache incoherency is a feature of the ARM native environment, this does not occur if the ARM binary is emulated for execution on a non-ARM-based architecture (e.g., x86 architecture). That is, EmuID can detect the cross-architecture emulation engine by checking that it does not run in the ARM native environment.

5.3. Evaluation on real mobile devices

We conducted another experiment to prove that EmuID also has no false-negatives by running it on ARM-based devices without emulation. In order to test EmuID on a wide variety of devices, we opted for the Amazon Device Farm service (AWS, 2020). We embedded EmuID code into an app through the *Java Native Interface (JNI)* and ran it on a total of 155 devices (all available device types in Amazon Device Farm) in addition to the four devices that we physically own. The reason why we ran such a thorough false-negative test is that there exist many ARM processor implementations. Also, a non-negligible portion of the processor architecture can be modified by the manufacturer due to the licensing practice of the ARM architecture. Fig. 10a shows the distributions of the manufacturer of the Android devices, while Fig. 10b shows the distributions of the installed operating systems on the Android devices.

Table 2 shows our experiment results. Each device is tested 1000 times with EmuID, and we did not find any false-positive results. Combined with our results from the experiment against emulation engines, this experiment again shows the accuracy of the EmuID detection method. Fig. 11 shows the time consumption for detecting the execution environment. The average time to detect is 0.14 ms. This shows that our method can be readily embedded into the software with almost no performance impact.

6. On the difficulty of mitigating EmuID

The emulation environment detection technique presented in this paper uses the hardware feature with differences that occur in the native environment but not in the emulation environment. The hardware feature is the incoherency between i-cache and d-cache. Our technique identifying the execution environment of the running application intentionally causes cache incoherency.

Table 2
False-positive test results from 1000 trials of detection on real devices.

Vendor	Device	Model	ARM Arch		OS	False-positive
			v7	v8		Rate
Raspberry Pi Foundation	RPI 2 Model B v1.1	BCM2836	✔		Ubuntu MATE 18.04.2	0%
	RPI 3 Model B v1.2	BCM2837		✔		0%
Linaro	Hikey 960	Kirin 960		✔	Lebuntu 16.04	0%
Arm	Juno r0	Juno r0		✔	Openembedded	0%
Samsung	Galaxy Note 9	SM-N960U1		✔	Android 8.1.0	0%
	Galaxy S9 (Unlocked)	SM-G960U1		✔	Android 8.0.0	0%
	Galaxy S9+ (Unlocked)	SM-G965U1		✔		0%
	Galaxy Tab S3 9.7"	SM-T820		✔		0%
	Galaxy Note8 (Unlocked)	SM-N950U1		✔	Android 7.1.1	0%
	Galaxy Note5 (AT&T)	SM-N9120A		✔	Android 7	0%
	Galaxy S6 (T-Mobile)	SM-G920T		✔		0%
	Galaxy S6 Edge	SM-G925F		✔		0%
	Galaxy S8 (T-Mobile)	SM-G950U		✔		0%
	Galaxy S8 Unlocked	SM-G950U1		✔		0%
	Galaxy S8+ (T-Mobile)	SM-G955U		✔		0%
	Galaxy Note5 SM-N920C	SM-N920C		✔	Android 6.0.1	0%
	Galaxy S5 (AT&T)	SM-G900A	✔			0%
	Galaxy S5 (Verizon)	SM-G900V	✔			0%
	Galaxy S6 (T-Mobile)	SM-G920T		✔		0%
	Galaxy S6 (Verizon)	SM-G920V		✔		0%
	Galaxy S6 Edge SM-G925F	SM-G925F		✔		0%
	Galaxy S6 SM-G920F	SM-G920F		✔		0%
	Galaxy S7 (AT&T)	SM-G930A		✔		0%
	Galaxy S7 Edge (AT&T)	SM-G935A		✔		0%
	Galaxy S7 Edge SM-G935F	SM-G935F		✔		0%
	Galaxy S7 SM-G930F	SM-G930F		✔		0%
	Galaxy Tab S2 9.7	SM-T813		✔		0%
	Galaxy Tab S2 8.0" (WiFi)	SM-T713		✔		0%
	Galaxy E5	SM-E500H	✔		Android 5.1.1	0%
	Galaxy Grand Prime 4G	SM-G531F	✔			0%
	Galaxy J5 4G	SM-J500F	✔			0%
	Galaxy Note5 (AT&T)	SM-N9120A		✔		0%
	Galaxy Note5 (T-Mobile)	SM-N920T		✔		0%
	Galaxy S6 Edge+ (AT&T)	SM-G928A		✔		0%
	Galaxy S6 Edge+ (T-Mobile)	SM-G928T		✔		0%
	Galaxy Tab S2 8.0" (WiFi)	SM-T710	✔			0%
	Galaxy A5	SM-A500F	✔		Android 5.0.2	0%
	Galaxy S6 (Verizon)	SM-G920V		✔		0%
	Galaxy S6 Edge	SM-G925F		✔		0%
	Galaxy S6 Edge (Verizon)	SM-G925V		✔		0%
	Galaxy Tab 4 10.1" (WiFi)	SM-T530NU	✔			0%
	Galaxy Note 4 (AT&T)	SM-N910A	✔		Android 5.0.1	0%
	Galaxy Note 4 (Verizon)	SM-N910V	✔			0%
	Galaxy Note 4 SM-N910H	SM-N910H	✔			0%
	Galaxy S4 (AT&T)	SGH-I337	✔			0%
	Galaxy S4 (Verizon)	SCH-I545	✔			0%
	Galaxy S4(Unlocked)	GT-I9500	✔			0%
	Galaxy Note 3 (Sprint)	SM-N900P	✔		Android 5	0%
	Galaxy Note 3 (T-Mobile)	SM-N900T	✔			0%
	Galaxy E7	SM-E7000	✔		Android 4.4.4	0%
	Galaxy Grand Neo Plus	GT-I9060I	✔			0%
	Galaxy Grand Prime Duos	SM-G530H	✔			0%
	Galaxy J1 Ace	SM-J110H	✔			0%
	Galaxy J1 Duos	SM-J100H	✔			0%
	Galaxy Note 3 (AT&T)	SM-N900A	✔			0%
	Galaxy Note 3 (Verizon)	SM-N900V	✔			0%
	Galaxy Note 4 (AT&T)	SM-N910A	✔			0%
	Galaxy Note 4 (Sprint)	SM-N910P	✔			0%
	Galaxy Note 4 (T-Mobile)	SM-N910T	✔			0%
	Galaxy Note 4 (Verizon)	SM-N910V	✔			0%
	Galaxy S DUOS 3	SM-G316HU	✔			0%
	Galaxy S4 (AT&T)	SGH-I337	✔			0%
	Galaxy S4 (T-Mobile)	SGH-M919	✔			0%
	Galaxy S5 (AT&T)	SM-G900A	✔			0%
	Galaxy S5 (Verizon)	SM-G900V	✔			0%
	Galaxy Tab 3 7.0" (T-Mobile)	SM-T217T	✔			0%
	Galaxy Grand 2	SM-G7102	✔		Android 4.4.2	0%
	Galaxy Light (MetroPCS)	SGH-T399N	✔			0%
	Galaxy Note 2 (AT&T)	SGH-I317	✔			0%
	Galaxy Note 2 (Verizon)	SCH-I605	✔			0%

(*continued on next page*)

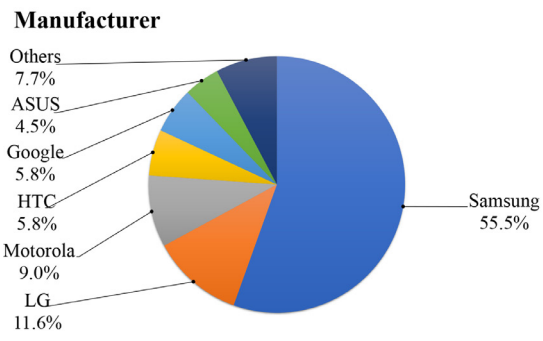
Table 2 (*continued*)

Vendor	Device	Model	ARM Arch		OS	False-positive
			v7	v8		Rate
	Galaxy Note 3 (AT&T)	SM-N900A	✔			0%
	Galaxy S3 (Verizon)	SCH-I535	✔			0%
	Galaxy S4 (AT&T)	SGH-I337	✔			0%
	Galaxy S4 (US Cellular)	SCH-R970	✔			0%
	Galaxy S4 (Verizon)	SCH-I545	✔			0%
	Galaxy S4 Active (AT&T)	SGH-I537	✔			0%
	Galaxy S4 mini (Verizon)	SCH-I435	✔			0%
	Galaxy S4 Mini GT-I9195	GT-I9195	✔			0%
	Galaxy S5 (AT&T)	SM-G900A	✔			0%
	Galaxy S5 (T-Mobile)	SM-G900T	✔			0%
	Galaxy S5 Active (AT&T)	SM-G870A	✔			0%
	Galaxy Tab 4 10.1" (WiFi)	SM-T530NU	✔			0%
	Galaxy Tab 4 7.0" (WiFi)	SM-T230NU	✔			0%
	Galaxy Note 2 (AT&T)	SGH-I317	✔		Android 4.3	0%
	Galaxy S3 (AT&T)	SGH-I747	✔			0%
	Galaxy S3 (T-Mobile)	SGH-T999	✔			0%
	Galaxy S3 (Verizon)	SCH-I535	✔			0%
	Galaxy S3 LTE (T-Mobile)	SGH-T999L	✔			0%
	Nexus 10 (WiFi)	GT-P8110	✔			0%
	Galaxy Tab 3 Lite 7.0" (WiFi)	SM-T110	✔		Android 4.2.2	0%
LG	G7 ThinQ	LM-G710		✔	Android 8.0.0	0%
LG	V20 (AT&T)	LG-H910		✔	Android 7	0%
	V20 (T-Mobile)	LG-H918		✔		0%
	V20 (Verizon)	VS995		✔		0%
	G5 (T-Mobile)	LG-H830		✔	Android 6.0.1	0%
	Nexus 5	D820	✔		Android 6	0%
	G3 (AT&T)	D850	✔		Android 5.0.1	0%
	Nexus 5	D820	✔			0%
	Nexus 4	E960	✔		Android 4.4.3	0%
	G Pad 7.0 (AT&T)	V410	✔		Android 4.4.2	0%
	G2 (AT&T)	D800	✔			0%
	G2 (T-Mobile)	D801	✔			0%
	G3 (AT&T)	D850	✔			0%
	G3 (T-Mobile)	D851	✔			0%
	G3 (Verizon)	VS985	✔			0%
	Nexus 5	D820	✔			0%
	Optimus L70 (MetroPCS)	MS323	✔			0%
	G Flex (AT&T)	D950	✔			0%
Motorola	Moto G 4	Moto G (4)	✔		Android 7	0%
	Nexus 6	XT1103	✔			0%
	Moto G - 2nd Gen	XT1064	✔		Android 6	0%
	Moto G - 3rd Gen	MotoG3	✔			0%
	Nexus 6	XT1103	✔			0%
	DROID Turbo 2 (Verizon)	XT1585		✔	Android 5.1.1	0%
	DROID Turbo (Verizon)	XT1254	✔		Android 5.1	0%
	Moto E - 2nd Gen	XT1511	✔			0%
	Moto X - 2nd Gen (Verizon)	XT1096	✔			0%
	Nexus 6	XT1103	✔			0%
	DROID Ultra (Verizon)	XT1080	✔		Android 4.4.4	0%
	Moto G (AT&T)	XT1045	✔			0%
	DROID RAZR HD (Verizon)	XT926	✔		Android 4.4.2	0%
	DROID RAZR M (Verizon)	XT907	✔			0%
HTC	U11	HTC U11		✔	Android 7.1.1	0%
	One A9 (Unlocked)	HTCOne A9		✔	Android 6.0.1	0%
	One M9 (AT&T)	6735A		✔	Android 5.0.2	0%
	One M9 (Verizon)	HTC6535LVW		✔		0%
	One M8 (AT&T)	6268A	✔		Android 4.4.4	0%
	One M8 (Verizon)	HTC6525LVW	✔			0%
	One M7 (AT&T)	6096A	✔		Android 4.4.2	0%
	One M8 (AT&T)	6268A	✔			0%
	One M8 (Verizon)	HTC6525LVW	✔			0%
Google	Pixel 2	Google Pixel 2		✔	Android 9	0%
	Pixel 2 XL	Google Pixel 2 XL		✔		0%
	Pixel 2	Google Pixel 2		✔	Android 8.1.0	0%
	Pixel	Pixel		✔	Android 8.0.0	0%
	Pixel XL	Pixel XL		✔		0%

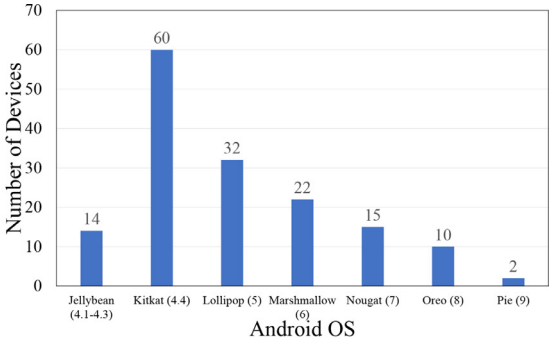
(*continued on next page*)

Table 2 (*continued*)

Vendor	Device	Model	ARM Arch		OS	False-positive
			v7	v8		Rate
	Pixel 2	Google Pixel 2		✔		0%
	Pixel 2 XL	Google Pixel 2 XL		✔		0%
	Pixel	Pixel		✔	Android 7.1.2	0%
	Pixel XL	Pixel XL		✔		0%
ASUS	Nexus 7 - 2nd Gen (WiFi)	ME571K	✔		Android 6	0%
	Nexus 7 - 2nd Gen (WiFi)	ME571K	✔		Android 5.0.1	0%
	Nexus 7 - 2nd Gen (WiFi)	ME571K	✔		Android 4.4.4	0%
	Nexus 7 - 2nd Gen (WiFi)	ME571K	✔		Android 4.4.2	0%
	Nexus 7 - 2nd Gen (WiFi)	ME571K	✔		Android 4.3.1	0%
	Nexus 7 - 1st Gen (WiFi)	ME370T	✔		Android 4.2.1	0%
	Nexus 7 - 1st Gen (WiFi)	ME370T	✔		Android 4.2	0%
Amazon	Fire HD 7 (2014)	SQ46CW	✔		Android 4.4.3	0%
	Kindle Fire HDX 7 (2013)	C9R6QM	✔			0%
	Fire Phone	SD4930UR	✔		Android 4.2.2	0%
Huawei	M8	HUAWEI NXT-L29		✔	Android 6	0%
	P9	EVA-L09		✔		0%
	Ascend Mate 7	MT7-L09	✔		Android 4.4.2	0%
Sony	Xperia Z4 Tablet	SGP712		✔	Android 5.0.2	0%
	Xperia Z3	D6616	✔		Android 4.4.4	0%
	Xperia Z1 Compact	D5503	✔		Android 4.3	0%
Intex	Aqua Y2 Pro	Aqua Y2 Pro	✔		Android 4.4.2	0%
Oppo	Find 7a	X9006	✔		Android 4.3	0%
Wiko	Rainbow 4G	RAINBOW 4G	✔		Android 4.4.2	0%



(a) Manufacturer of 155 real android devices tested at Amazon Device Farm. The real mobile devices used for evaluation are Samsung 86 (55.5%), LG 12 (11.6%), Motorola 14 (9.0%), HTC 9 (5.8%), Google 9 (5.8%), ASUS 7 Large (4.5%), Others 12 (7.7%).



(b) Android OS of 155 real android devices tested at Amazon Device Farm. 14 devices with Jellybean (4.1-4.3) OS, 60 devices with Kitkat (4.4) OS, 32 devices with Lollipop (5) OS, 22 devices with Marshmallow (6) OS, 22 Nougat (7)), 15 devices with OS, 10 devices with Oreo (8) as OS, and 2 devices with Pie (9) as OS were used for the test.

Fig. 10. Manufacturer and OS version distributions of tested devices.

Self-modifying code is suitable as the code that occurs cache incoherency. When our detection technique utilizing self-modifying code works, abnormal execution results due to cache incoherency occur in the native environment, and normal execution occurs without cache incoherency in the emulation environment. Since these results are based on architectural features, it is difficult to be bypassed with simple modifications, unlike heuristic techniques. In this section, we discsuss why it is non-trivial to amend the emulation engines to mitigate EmuID through a few plausible but infeasible theoretic mitigation methods.

6.1. Detecting and nullifying *EmuID* through manual analysis

In EmuID deployment scenarios, we assume that EmuID will often be used in conjunction with software obfuscation techniques (Themida, 2021). The program code itself is encrypted or obfuscated in arbitrary ways and only reveals its behavior during runtime. As such, EmuID will be hidden among the already obfuscated program code. In addition, our proof-concept shown in Fig. 9, can be arbitrarily transformed such that it evades signatures-based detection mechanisms as often seen in polymorphic malware. These efforts can substantially raise the bar for the reverse engineers since obfuscated programs often require dynamic analysis, and EmuID makes such a method rather difficult.

However, skilled reverse engineers can eventually disarm EmuID code hidden in the program with sufficient efforts, as with any other anti-emulation features. With the knowledge of the EmuID's mechanism, the reverse engineers can locate and eliminate EmuID code from the program. Hence, EmuID cannot provide deterministic prevention of manual analysis but rather brings additional hurdles in the analysis process. Nevertheless, EmuID leverages the unique characteristics of the ARM architecture. Therefore, fundamentally mitigating EmuID through rectifying the emulation engines would be rather difficult unlike the existing signature-based emulation detection schemes (Falcón and Riva, 2012; Hron and Jermář, 2014; Jang et al., 2019; Jing et al., 2014; Kirsch et al.,

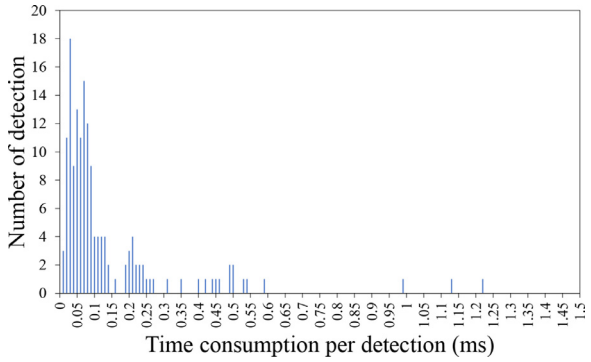


Fig. 11. Time consumption for detection on 155 real android devices of Amazon Device Farm. The average time to detect is 0.14 ms. The shortest time is 0.012 ms, and the longest time is 1.224 ms.

2018; Li and Li, 2014; Petsas et al., 2014; Polino et al., 2017; Raffetseder et al., 2007; Sun et al., 2016). We will discuss this point later in this section (Sections 6.3 and 6.4).

6.2. Compatibility issues with W^X Policy

EmuID has a reliance on the self-modifying code and hence violates the W^X policy. Also, as we assume that software obfuscation methods are used in combination, violation of the W^X policy is inevitable.

When EmuID-protected program is run inside emulation environments for the purpose of analysis, the violation of the W^X policy is usually tolerated. This is because the emulation software themselves require JIT compilation to translate or instrument the emulated software, and this means that system-wide enforcement of W^X is infeasible. The emulation engines can also enforce the W^X policy during emulation. However, many obfuscated or protected software does employ self-modifying code, and the reverse engineer has no option but to allow such behavior.

However, running EmuID-protected program in native execution environments for benign use can bring compatibility issues. Some modern systems may strictly enforce the W^X policy, thereby rendering EmuID-protected program unable to run on the system. This is a limitation of EmuID as well as many software obfuscation methods that include self-modifying code (Themida, 2021; upx, 2021).

6.3. Modifying code translation granularity

Altering the translation granularity of software emulation engines may be considered a possible mitigation for the EmuID detection method. Since EmuID takes advantage of the artifact that happens during the code translation process that copies and translates the next basic block to be executed. However, a close look into the mechanism by which the translation code cache is maintained reveals the tentative mitigation approach's inapplicability. An emulation engine has to meet the following requirements to mimic the architectural characteristics that appear due to the cache lines:

One tentative solution would be to redesign software emulation engines such that it fetches code at a cache-line granularity. This solution introduces a few non-trivial problems. First, this requires heavy modifications to the emulation engines at the fundamental level. Emulation engines (Bellard, 2005; Bruening et al., 2003), simply assume basic block granularity in all components of its implementation. This is simply because the basic block by basic block translation is the most efficient and intuitive. Therefore, changing the translation granularity might mean reconsidering the emulation design from scratch.

Second, this solution would introduce significant performance overhead. If a single basic block spans multiple cache lines, the basic block would have to be split into multiple code cache. This would increase the amount of control flow transfers among the code caches, hurting the cache locality. When multiple basic blocks reside in the same cache line, the emulation engine must perform translation of all the basics blocks that are not necessarily going to be executed. Overall, this tentative solution would introduce a performance overhead as well as software complexity.

Third, modifying translation code cache generation and maintenance for EmuID mitigation would also render existing code cache optimization incompatible. For instance, *basic block linking* and *trace caching* must be either discarded or redesigned for the modification. Basic block linking stitches two basic blocks together to avoid a call to emulation dispatcher. Furthermore, frequently executed sequences of basic blocks are chained into a *trace* for additional performance boost (Dynamorio system details, 2020). Attempts to change the granularity of code fetch and translate are likely to conflict with these general optimizations strategies.

For the reasons we explained above, we argue that changing the translation granularity is not realistic mitigation for EmuID. In all, modifying the translation granularity solely to mitigate EmuID might be possible. However, the amount of effort that is required and mounting performance problems make the approach infeasible.

6.4. Emulating native cache behavior

A robust mitigation to EmuID and its variants would be to modify the emulation engine or implement a set of translation rules to maintain a virtual set of L1 caches throughout the program emulation. First and foremost, the emulation engine must be monitoring all changes to executable pages. To achieve this goal, the engine must watch the page permission changes as well as the creation of new pages that introduce RWX pages. Then, it includes tracing of all memory accesses, cache-flushing instructions, and a *native-like* cache eviction policy. This is because EmuID is crafted to leave a stale value only in the i-cache; the only way to capture the stale value or incoherency is to closly follow all possible cache content changes to calculate the *current* i-cache content of the currently executing cache lines.

We found that implementing the above mitigation would be a daunting challenge to emulation engine developers. Cache profiling through emulation engines havs been developed and used in practice (Cache simulator, 2020). However, they currently only report cache hit rates for the caches and the addresses of the cache lines, but not the contents of the cache lines. For detection of EmuID, the cache profiling tool must be modified to also track the contents of the cache lines. More importantly, the performance overhead of the tool rules out the cache profiling tool as possible mitigation for EmuID. The documentation of the profiling tool explains that the tool is too slow to profile an entire application since the performance overhead is about 500 times that of the native execution (Cache simulator, 2020). Therefore, mitigating EmuID through an always maintained virtual cache is also not feasible.

7. Conclusion

In this paper, we presented EmuID, which takes advantage of the characteristics of the ARM architecture's cache behavior to detect the presence of emulation. We provided an in-depth analysis of our detection method and how it causes the native and emulated execution environments to have different cache behaviors, which is utilized for the detection method. We showed that our

method is accurate and agnostic to implementations of emulators by testing it on well-known software emulation engines such as DBI (Valgrind, DynamoRIO) for the ARM architecture and emulator (QEMU) for the ARM architecture and x86 architecture, and also confirmed that it has no false positives on 155 ARM-based devices.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2020R1A2C2101134, NRF-2020R1C1C1011980) and the Institute for Information and communication Technology Promotion (IITP) grant (No. IITP-2017-0-01889, IITP-2020-0-00666, IITP-2021-0-01366, IITP-2021-0-01587). This work was also supported by the Office of Naval Research (ONR) through Award (N00014-18-1-2661).

Appendix A. First Appendix

The EmuID code for 64-bit ARM architecture is illustrated in Fig. 9, but shell code of this is longer then the cache line size in 32-bit ARM architecture. Thus, we implemented the EmuID code for 32-bit ARM architecture using Thumb instruction set that is consists of 16-bit instructions. Fig. A.1 is EmuID code written for the 32-bit ARM architecture.

The code from line 7 to line 21 is the launcher (L), and from line 31 is the detector (D). The branch instruction in line 21 makes the L and the D be different basic blocks. The L first change the instruction mode from the standard ARM to Thumb (line 7–8 in Fig. A.1). After then, the L sets the registers in line 12–14. The register r0 is used as a Pointer to Code Position. The register r1 and r2 are used as the temp registers. The register r3 is used as the xor key and the loop counter. After the register values are set, the L modifies the D, which has initially the bx lr instruction. Line 15–21 contain the contents of the loop for xor. In this loop, the bx lr instructions is of D will be unpacked through xor operation with the key. The bx lr instruction is modified to mov r0, sp instruction. When L finishes executing, now D' will start executing. In case of native execution, a stale copy of D in i-cache will execute and EmuID code will return immediately with changing instruction mode to standard ARM. On the other hand, in emulation execution, D'(or more precisely Dt's) will execute undefined instruction after executing mov r0, sp and get a illegal instruction fault.

```
1
2  // r0 : Pointer to Code Position
3  // r1 : Temp Register for XOR
4  // r2 : Temp Register for loop counting
5  // r3 : XOR Key, Loop Counter
6
7  0x00000000:  01 10 8F E2    add   r1, pc, #1
8  0x00000004:  11 FF 2F E1    bx    r1
9
10 // Change to Thumb mode
11
12 0x00000008:  92 1A    subs r2, r2, r2
13 0x0000000a:  02 23    movs r3, #2
14 0x0000000c:  03 A0    adr  r0, #0xc
15 0x0000000e:  01 68    ldr  r1, [r0]
16 0x00000010:  59 40    eors r1, r3
17 0x00000012:  01 60    str  r1, [r0]
18 0x00000014:  01 30    adds r0, #1
19 0x00000016:  01 3B    subs r3, #1
20 0x00000018:  9A 42    cmp  r2, r3
21 0x0000001a:  F8 D1    bne  #0xe
22
23 // XOR every byte with key beyond this point
24 // Convert "bx lr" to "mov r0, sp"
25 // xor(0x70,0x2) = 0x68
26 // xor(0x47,0x1) = 0x46
27 // \x68\x46 is "mov r0, sp" in Thumb mode
28
29
30 // Detector (D)
31 0x0000001c:  70 47    bx lr // RET and Change to ARM mode
32 // In the native, returns to the original caller
33 // In the emulated, the following invaild instruction is
       executed.
34
35 0x0000001e:  ff ff    (Invaild instruction)
36 ...
```

Fig. A.1. Self-Modifying code of EmuID for 32-bit ARM architecture.

References

Aarch64 port, 2020. https://github.com/DynamoRIO/dynamorio/wiki/AArch64-Port.
Anton, C., Mark, H., Ray, H., Chris, R., Norman, R., 1998. Fx! 32–a profile-directed binary translator. IEEE Micro 18 (2), 56–64.
Aws device farm, 2020. https://aws.amazon.com/device-farm/.
Bellard, F., 2005. QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track, Vol. 41, p. 46.
Bochs: The open source ia-32 emulation project, 2020. http://bochs.sourceforge.net.
Brengel, M., Backes, M., Rossow, C., 2016. Detecting hardware-assisted virtualization. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 207–227.
Bruening, D., Garnett, T., Amarasinghe, S., 2003. An infrastructure for adaptive dynamic optimization. In: International Symposium on Code Generation and Optimization, 2003. CGO 2003. IEEE, pp. 265–275.
Bruening, D., Zhao, Q., Amarasinghe, S., 2012. Transparent dynamic instrumentation. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments, pp. 133–144.
Buck, B., Hollingsworth, J.K., 2000. An API for runtime code patching. Int. J. High Perform. Comput. Appl. 14 (4), 317–329.
Cache simulator, 2020. https://dynamorio.org/dynamorio_docs/page_drcachesim.html.
D'Elia, D.C., Coppa, E., Nicchi, S., Palmaro, F., Cavallaro, L., 2019. SoK: using dynamic binary instrumentation for security (and how you may get caught red handed). In: Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, pp. 15–27.
Dynamorio system details, 2020. https://dynamorio.org/dynamorio_docs/overview.html.
Falcón, F., Riva, N., 2012. Dynamic binary instrumentation frameworks: i know youre there spying on me. In: Reverse Engineering Conference.
Ferrie, P., 2007. Attacks on more virtual machine emulators. Symantec Technol. Exch. 55, 369.
Garfinkel, T., Adams, K., Warfield, A., Franklin, J., 2007. Compatibility is not transparency: VMM detection myths and realities. HotOS.
Garfinkel, T., Adams, K., Warfield, A., Franklin, J., 2007. Compatibility is not transparency: VMM detection myths and realities. HotOS.
Garfinkel, T., Rosenblum, M., et al., 2003. A virtual machine introspection based architecture for intrusion detection. In: Ndss, Vol. 3. Citeseer, pp. 191–206.
Hazelwood, K., Klauser, A., 2006. A dynamic binary instrumentation engine for the arm architecture. In: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp. 261–270.
Hron, M., Jermář, J., 2014. SafeMachine: Malware Needs Love, Too. Virus Bulletin.
Jacob, B., 2013. Caches and self modifying code. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/caches-and-self-modifying-code, Accessed: 2020-09.
Jang, D., Jeong, Y., Lee, S., Park, M., Kwak, K., Kim, D., Kang, B.B., 2019. Rethinking anti-emulation techniques for large-scale software deployment. Comput. Secur. 83, 182–200.
Jing, Y., Zhao, Z., Ahn, G.-J., Hu, H., 2014. Morpheus: automatically generating heuristics to detect android emulators. In: Proceedings of the 30th Annual Computer Security Applications Conference, pp. 216–225.
Karl Trygve Kalleberg, O.A.V.R., 2016. Testing Interoperability with Closed-Source Software Through Scriptable Diplomacy. FOSDEM.
Kirsch, J., Zhechev, Z., Bierbaumer, B., Kittel, T., 2018. Pwin–pwning intel pin: why DBI is unsuitable for security applications. In: European Symposium on Research in Computer Security. Springer, pp. 363–382.
Kvm, 2020. http://linux-kvm.org/.
Li, X., Li, K., 2014. Defeating the Transparency Features of Dynamic Binary Instrumentation. BlackHat US.
Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K., 2005. Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Not. 40 (6), 190–200.
Mulliner, C., Oberheide, J., Robertson, W., Kirda, E., 2013. PatchDroid: scalable third–party security patches for android devices. In: Proceedings of the 29th Annual Computer Security Applications Conference, pp. 259–268.

Nethercote, N., Seward, J., 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN Not. 42 (6), 89–100.
Oracle vm virtualbox, 2020. https://www.virtualbox.org/.
Payer, M., Gross, T.R., 2011. Fine-grained user-space security through virtualization. ACM SIGPLAN Not. 46 (7), 157–168.
Pék, G., Bencsáth, B., Buttyán, L., 2011. nether: In-guest detection of out-of-the-guest malware analyzers. In: Proceedings of the Fourth European Workshop on System Security, pp. 1–6.
Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S., 2014. Rage against the virtual machine: hindering dynamic analysis of android malware. In: Proceedings of the Seventh European Workshop on System Security, pp. 1–6.
Polino, M., Continella, A., Mariani, S., D'Alessio, S., Fontana, L., Gritti, F., Zanero, S., 2017. Measuring and defeating anti-instrumentation-equipped malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 73–96.
Qemu internals, 2012. https://qemu.weilnetz.de/w64/2012/2012-06-28/qemu-tech.html.
Quarkslab, 2019. QDBI. https://qbdi.quarkslab.com/.
Quynh, N.A., 2018. SKORPIO: Advanced Binary Instrumentation Framework. OPCDE.
Raffetseder, T., Kruegel, C., Kirda, E., 2007. Detecting system emulators. In: International Conference on Information Security. Springer, pp. 1–18.
Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J.W., Soffa, M.L., 2003. Retargetable and reconfigurable software dynamic translation. In: International Symposium on Code Generation and Optimization, 2003. CGO 2003. IEEE, pp. 36–47.
Sun, K., Li, X., Ou, Y., 2016. Break Out of The Truman Show: Active Detection and Escape of Dynamic Binary Instrumentation. Black Hat Asia.
Themida, 2021. https://www.oreans.com/Themida.php.
Thompson, C., Huntley, M., Link, C., 2010. Virtualization Detection: New Strategies and Their Effectiveness. University of Minnesota. (unpublished)
upx, 2021. https://upx.github.io/.
Virtualpc, 2020. www.microsoft.com/windows/virtual-pc/.
Vmware, 2020. http://www.vmware.com/.
Zyngie, M., 2015. Arm: Caches That Give You Enough Rope to Shoot yourself in the Foot by Marc Zyngier. KVM Forum 2015.

Yeseul Choi received the B.S. degree in Computer Science from Handong Global University in 2015. She also received the M.S. in the Graduate School of Information Security at Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2016. She is currently working toward the Ph.D. degree at the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). Her research interest includes software exploitation mitigation computer system security, anti-emulation, trusted execution environments (TEEs).

Yunjong Jeong is currently a Ph.D candidate at Graduate School of Information Security from Korea Advanced Institute of Science and Technology (KAIST), South Korea, and also received the M.S. and B.S. from KAIST. His research interest includes systems and software security, in particular trusted execution environments (TEEs), securing cloud workload and applied cryptography.

Daehee Jang received B.S. degree in Computer Engineering at Hanyang University, South Korea, in 2012. He received M.S. and Ph.D. degree in Information Security at Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2014 and 2019. Afterward, he joined as a postdoctoral researcher at Georgia Tech, USA until the end of 2020. Currently, he is an assistant professor at Sungshin W. University Department of Convergence Security Engineering since 2021. Notably, he participated in various hacking competitions (including the DEFCON CTF finals) and won several awards. Also, he is the founder of pwnable.kr wargame. His research interests mainly include software vulnerability attack and mitigation, fuzzing, and web security.

Hojoon Lee is currently an assistant professor at Department of Computer Science and Engineering at Sungkyunkwan University since September, 2019. Prior to his current position, he spent one year as a postdoctoral researcher at CISPA under supervison of Prof. Michael Backes. He recevied Ph.D from KAIST in 2018 advised by Prof. Brent Byunghoon Kang and his B.S. from The University of Texas at Austin. His main research interests lie in retrofitting security in computing systems against today's advanced threats. His research interests include but not limited to Operating System Security, Trusted Execution Environments, Program Analysis, Software Security, and Secure AI Computation in Cloud.

Brent Byunghoon Kang received the B.S. degree from Seoul National University, the M.S. degree from the University of Maryland at College Park, and the Ph.D. degree in computer science from the University of California at Berkeley. He has been an Associate Professor with George Mason University. He is currently a Professor with the Graduate School of Information Security, Korea Advanced Institute of Science and Technology (KAIST). He has been working in the field of systems security area, including botnet defense, OS kernel integrity monitors, trusted execution environment, and hardware-assisted security. He is a member of USENIX and ACM.