January 2020

Securing Arm Platform: From Software-Based To Hardware-Based Approaches

Zhenyu Ning
Wayne State University

SECURING ARM PLATFORM: FROM SOFTWARE-BASED TO HARDWARE-BASED APPROACHES

by

ZHENYU NING

DISSERTATION

Submitted to the Graduate School,

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2020

MAJOR: COMPUTER SCIENCE

Approved By:

_____

Advisor Date

DEDICATION

To my wife and all my family.

ACKNOWLEDGEMENTS

I wish to express my appreciation to those who supported and encouraged me in one way or another during the last five years.

First of all, I would like to express my gratefulness to my advisor, Dr. Fengwei Zhang, for his valuable guidance and continuous support during my Ph.D. study at Wayne State University. Dr. Zhang led me into the way of academic research and showed me the essentials of a researcher. His enthusiasm for research has encouraged me to achieve the degree. During the tough times in the Ph.D. pursuit, he showed me great patient and offered numerous hours to discuss with me, without which I cannot succeed.

I am also very grateful to Prof. Weisong Shi, Prof. Dongxiao Zhu, and Prof. Anyi Liu for serving as my dissertation committee and giving valuable suggestions on the dissertation. Your attitude on research helps me not only in the dissertation but also in my career.

Moreover, I would like to thank all the group members in COMPASS lab. Especially for Dr. Saeid Mofrad, Dr. Lei Zhou, and Dr. Jinghui Liao, we have had many brilliant ideas and exciting discussions.

Also, I deeply appreciate the support and love from my wife Qian Jiang. You give me light in the darkness and always make me hopeful. Thanks to my mother Weiping Zhu and my father Guoqiang Ning, I really hope you could see it.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

ARM architecture has been wildly adopted in smart mobile phones and Internet of Things (IoT). In recent years, smart mobile phones and IoT devices become prevalent and important to our daily life. Unfortunately, these devices suffers from a variety of malware threats and internet attacks. According to the latest McAfee threats report, more than 1.5 million new malware samples on mobile device has been detected during the third quarter of 2018 [161]. To defend against these malware attacks, researchers need to analyze these samples to understand their behavior so that effective defenses can be developed.

As the most representative platform for mobile phones and IoT devices, Android dominate the mobile operate system market with more than 75% market share by the end of 2018 [238]. The dominative market also attracts the attentions of malicious developers and security researchers. To avoid of being detected by the analysis systems designed by the security researchers, the malicious developers play with the special Java virtual machine deployed in Android and adopt various techniques to hide the malicious behavior in the application [5, 42, 48, 113, 114, 151, 198, 248]. Specifically, these techniques aim to obfuscate the bytecode inside the application and mislead the analysis systems.

Other than the special techniques for Android, the traditional evasion trick helps the malware escape from being detected by the malware analysis system on ARM platforms. For example, some of the malware analysis systems on ARM platforms [73, 245, 277] are based on emulation or virtualization technology, and a series of anti-emulation and anti-virtualization techniques [131, 193, 258] have been developed to challenge them. To address this challenge, researchers study the malware on bare-metal devices via modify-

ing the system software [85, 194, 244, 294] or leveraging OS APIs [55, 295] to monitor the runtime behavior of malware. Although bare-metal based approaches eliminate the detection of the emulator or hypervisor, the artifacts introduced by the analysis tool itself are still detectable by malware. Moreover, privileged malware can even manipulate the analysis tool since they run in the same environment. How to build a transparent malware analysis system on ARM platform is still a challenging problem.

This transparency problem has been well studied in the traditional x86 architecture, and milestones have been made from emulation-based analysis systems [15, 234] to hardware-assisted virtualization analysis systems [76, 77, 147], and then to bare-metal analysis systems [140, 141, 236, 285]. As the state-of-the-art solution to the transparency problem on x86 architecture, [285] provide us a insight that the hardware-based design brings a better transparency for the analysis system.

The goal of our work it to provide a practical solution for the security-related problems on the ARM platforms. As the first attempt, we design a novel program transformation system that reveals the hidden code in Android applications to analyzable pattern via instruction-level extracting and reassembling. With this transformation, the hidden code protected by the aforementioned evasion techniques on Android (i.e., packing, reflection, dynamic loading and self-modifying) would be detectable for existing static analysis tools. Our system collects bytecode and data when they are executed and accessed, and reassembles the collected result into a valid DEX file for static analysis tools. One of the key challenges in our system is to reassemble the instructions into a valid and accurate DEX file. Hence, we design a novel reassembling approach to construct the entire executed control flows including self-modifying code. Additionally, we implement the first prototype of

force execution on Android and use it as our code coverage improvement module. Since we extract all executed instructions, our system is able to uncover the malicious behavior of the applications equipped with Android-specific evasion technique.

Next, to handle the traditional evasion techniques, we present a transparent malware analysis framework on ARM platform by leveraging hardware features including TrustZone technology, Performance Monitoring Unit (PMU), and Embedded Trace Macrocell (ETM). We implement a prototype of the framework that embodies a trace subsystem with different tracing granularities and a debug subsystem with a GDB-like debugging protocol on ARM Juno development board. In the prototype, we also protect the system registers via hardware traps and memory protection to keep the analysis system transparent to the target application.

Due to the heavy use of the hardware debugging features, we also dig into the ARM debugging architecture to acquire a comprehensive understanding of the debugging features. Although the debugging architecture has been presented for years, its security is under-examined by the community since it normally requires physical access to use these features. However, the real security aspects of the debugging architecture remains unclear. During the analysis, we find that physical access is not actually required to make use of the hardware debugging features. Consequently, we summarize a series of the security implications that is caused by the assumption of physical access. By exploiting these implications, we craft a novel attack scenario that works on a processor running in a low-privileged mode and accesses the high-privileged content of the system without restriction via the misusing the hardware debugging features.

The revealed privilege escalation vulnerability has raised our concern on the security

of the Trusted Execution Environment (TEE) and Cyber-physical Systems (CPS). For the security of TEE, we conduct an analysis on the widely deployed TEEs and summarize the challenges in securing these TEEs. A study of the deploying the TEEs on edge platform is also presented. For the security of CPS, we also perform an analysis on the real-world traffic signal systems to understand their security problems.

CHAPTER 2 BACKGROUND

In this section, we introduce the basic concepts used in this paper. First, we explain the Java virtual machines deployed in Android. Then, I describe the design of ARM TrustZone and trusted firmware. The ARM debugging architecture is also explained.

2.1 Dalvik, Android Runtime, and Android Java Bytecode

Dalvik is a special Java virtual machine running in the Android system. It is used to interpret Android specified bytecode format since the first release of Android. To improve the performance, Google has introduced Just-In-Time (JIT) compilation and Ahead-Of-Time (AOT) compilation since Android 2.2 and Android 4.4, respectively. The JIT compilation continually compiles frequently executed bytecode slices into the machine code. As an upgrade, the AOT compilation compiles most bytecode in the application into the machine code during the installation. Dalvik equipped with AOT compilation is renamed to Android Runtime (ART). Since Android 5.0, Dalvik has been completely replaced by ART.

In both Dalvik and ART, the Java source code is compiled to Dalvik Executable (DEX) files which includes the bytecode for the Android Java virtual machine. The bytecode in DEX files is organized in units of methods. The minimum code unit for JIT and AOT compilation is a method, indicating that a single method cannot contain both bytecode and machine code. Methods such as constructors and abstract methods require the bytecode interpreter even in ART. Moreover, a single method or the entire ART can be configured to run in the interpreter mode.

The Java bytecode in Android is chained by instructions. Each instruction contains an opcode and arguments related to the opcode. The opcodes are different from the ones in

regular Java bytecode and the bit-length of an instruction varies according to the opcode. In the interpreter, instructions are listed in an array of 16-bit (2 bytes) units. An instruction occupies at least one unit with a maximum number of units up to five.

2.2 ARM TrustZone and Trusted Firmware

ARM TrustZone technology [31] introduces a hardware-assisted security concept that divides the execution environment into two isolated domains, i.e., secure domain and non-secure domain. Due to security concerns, the secure domain could access the resources (e.g., memory and registers) of the non-secure domain, but not vice versa. In ARMv8 architecture, the only way to switch from normal domain to secure domain is to trigger a secure exception [20], and the exception return instruction `eret` is used to switch back to the normal domain from the secure domain after the exception is handled.

ARM Trusted Firmware [30] (ATF) is an official implementation of secure domain provided by ARM, and it supports an array of hardware platforms and emulators. While entering the secure domain, the ATF saves the context of the normal domain and dispatches the secure exception to the corresponding exception handler. After the handler finishes the handling process, the ATF restores the context of the normal domain and switches back with `eret` instruction. ATF also provides a trusted boot path by authenticating the firmware image with several approaches like signatures and public keys.

2.3 ARM Debugging Architecture

The ARM architecture defines both invasive and non-invasive debugging features [18, 20]. The invasive debugging is defined as a debug process where a processor can be controlled and observed, whereas the non-invasive debugging involves observation only

without the control. The debugging features such as breakpoint and software stepping belong to the invasive debugging since they are used to halt the processor and modify its state, while the debugging features such as tracing (via the Embedded Trace Macrocell) and monitoring (via the Performance Monitor Unit) are non-invasive debugging.

The invasive debugging can be performed in two different modes: the halting-debug mode and the monitor-debug mode. In the halting-debug mode, the processor halts and enters the debug state when a debug event (e.g., a hardware breakpoint) occurs. In the debug state, the processor stops executing the instruction indicated by the program counter, and a debugger, either an on-chip component such as another processor or an off-chip component such as a JTAG debugger, can examine and modify the processor state via the Debug Access Port (DAP). In the monitor-debug mode, the processor takes a debug exception instead of halting when the debug events occur. A special piece of software, known as a monitor, can take control and alter the process state accordingly.

2.4 Advanced Transportation Controller and Roadside Cabinets

The traffic signals in the intersections are controlled by the Advanced Transportation Controller (ATC). The ATC makes logical decisions based on its inputs and configuration settings to implement traffic patterns. This configuration, based upon what is called the signal timing plan [257], holds parameters such as what duration to run which traffic patterns along with the minimum and maximum times to run the pattern.

According to the ATC standard [2] released by American Association of State Highway and Transportation Officials (AASHTO) [1], Institute of Transportation Engineers (ITE) [124], and National Electrical Manufacturers Association (NEMA) [171], the ATC

is built upon a Linux kernel with BusyBox integration, supporting a capable networking stack and access to most typical Linux shell operations such as FTP and SSH. On top of the kernel, the actual control logic is left to the individual software running in the ATC, and the municipalities may use different software according to their specific requirements and existing infrastructure.

The ATC is normally placed in a roadside cabinet. There are mainly two standards for the cabinet, i.e., the TS-2 standard [170] designed by NEMA and the Intelligent Transportation System (ITS) standard [115] developed by ITE.

The TS-2 Cabinet Standard [170] is a traffic signal cabinet standard that was initially commissioned by NEMA in 1998. The core feature of the modern TS-2 cabinet is its use of a single IBM SDLC serial bus for inter-device communications within the cabinet. The ITS Cabinet standard [115] is designed to supersede the NEMA TS-2 standard. By effectively using two serial buses, the ITS Cabinet maintains separation between the control plane of the traffic signal's relays and the supervisory bus shared between the traffic controller unit and fail-safe unit. Since the control planes (failure handling, signal control, environmental sensing) is separated into different buses, the congestion and latency on the bus are reduced.

2.4.1 Malfunction Management Unit and Cabinet Monitor Unit

As specified in the NEMA TS-2 Cabinet specification, the Malfunction Management Unit (MMU) is designed to accomplish the detection of, and response to, improper and conflicting signals. If an MMU detects that any monitoring parameter is out-of-range or in disagreement with the expectation, the MMU will override the control of the ATC, and the intersection is placed into a known-safe state called "conflict flash". Conflict flash is a state

in that all intersection operations are halted and individual traffic signal will be instructed to strobe their red lights. In order to return the operation of an intersection to a normal state, the MMU must be manually reset by a technician on-site.

The functionality of the Cabinet Monitor Unit (CMU) in the ITS cabinet is similar to the MMU in the TS-2 cabinet. The ITS Cabinet specification states that the minimum functionality of CMU is as least that provided by the NEMA TS-2 MMU. Additionally, the CMU offers enhanced monitoring and logging capabilities for items such as electrical voltages seen on cabinet peripherals, operating temperatures, and access controls.

CHAPTER 3 REASSEMBLEABLE BYTECODE EXTRACTION ON ANDROID

3.1 Introduction

For a better understanding of the malware behaviors on Android, a series of static analysis tools [33, 58, 98, 148, 265] and dynamic analysis tools [85, 244, 245, 277, 294] are designed. However, these tools suffer from some common disadvantages.

First, static analysis tools identify the malicious behavior of an application by investigating bytecode in Dalvik Executable (DEX) files, which is compiled from the Java source code and embedded in the Android Package (APK) file of the application. Due to the popular usage of the public packing platforms [5, 42, 114, 151, 198, 248], the original DEX file of the application may be encrypted and replaced by another shell DEX file, while the shell DEX file would decrypt the original DEX file and release it at runtime. In this case, static analysis tools are completely unarmed as they can only fetch the shell DEX file but not the encrypted original DEX file. Existing solutions [278, 293] to the packing technique assume that there is a point when all original code is unpacked in memory, which is not held with sophisticated adversaries [48, 113]. Moreover, the Java reflection and dynamic loading code are still a challenging task for the static analysis tools [33, 58, 98].

Second, although the dynamic analysis tools [85, 244, 245, 277, 294] do not suffer from the aforementioned techniques, they have their own drawbacks. The automatic dynamic taint flow analysis tools [85, 244, 294] cannot handle implicit taint flows while static analysis tools [148, 265] can solve them. Moreover, the huge performance overhead makes it difficult to implement a complicated analysis mechanism, so there is a trade-off between the accuracy and performance. Meantime, the code coverage problem also

threatens the accuracy of the dynamic analysis tools [245, 277, 294].

We present DexLego, a novel program transformation system that reveals the hidden code in Android applications to analyzable pattern via instruction-level extracting and reassembling. DexLego collects bytecode and data when they are executed and accessed, and reassembles the collected result into a valid DEX file for static analysis tools. Since we extract all executed instructions, our system is able to uncover the malicious behavior of the packed applications or malware with self-modifying code. One of the key challenges in DexLego is to reassemble the instructions into a valid and accurate DEX file. Hence, we design a novel reassembling approach to construct the entire executed control flows including self-modifying code. Additionally, we implement the first prototype of force execution on Android and use it as our code coverage improvement module.

Moreover, our system helps static analysis tools improve the analysis accuracy on reflection samples. The Java reflection obscures the control flows of the application by replacing the direct function call or field access with a call to the reflection library functions which take the name string of the function or field as parameter. Previous reflection solutions [45] and static analysis tools [33, 58, 98] on Android assume that the name strings of the reflectively invoked method and its declaring class are reachable. However, the name string can be encrypted in some cases [204] and the advanced malware could even use reflective method calls without involving any string parameter [83]. A solution on traditional Java platform [49] requires load-time instrumentation which is not supported in Android [33]. Thus, DexLego implements a similar idea in Android and replaces the reflective call with direct call.

We evaluate DexLego on real-world packed applications and DroidBench [83]. The

evaluation result shows DexLego successfully unpack and reconstruct the behavior of the applications. The F-measures (i.e., analysis accuracy) of FlowDroid [33], DroidSafe [98], and HornDroid [58] on DroidBench increase 33.3%, 31.1%, and 23.6%, respectively. Moreover, static analysis tools with the help of DexLego provide a better accuracy than existing dynamic analysis systems TaindDroid [85] and TaindART [244]. The code coverage experiments on open source samples from F-Droid [86] show that our force execution module helps to improve the coverage of dynamic analysis and increases the coverage of state-of-the-art fuzzing tool, Sapienz [158], from 32% to 82%.

3.2 Related Work

3.2.1 Static Analysis Tools

FlowDroid [33] is a static taint-analysis tool for Android applications, and it achieves a high accuracy by mitigating the gaps between lifecycle methods and callback methods. Amandroid [265] and IccTA [148] aim to resolve the implicit control flows during inter-component communication. EdgeMiner [59] links the callback methods with their registration methods to facilitate the static analysis tools in gaining more precise results. DroidSafe [98] implements a simplified model of the Android system and solves native code in the Android framework by manually analyzing the source code and writing stubs for them in Java. HornDroid [58] generates Horn clauses from the bytecode of application and performs both value-sensitive and flow-sensitive analysis on the clauses. HSOMiner [189] uses machine learning algorithms to discover the hidden sensitive operations by analyzing the branch instructions and their related conditional branches.

3.2.2 Dynamic Analysis Tools

DroidScope [277] provides an instrumentation tool to monitor the executed bytecode and native instructions to help analysts learn the malware manually. VetDroid [294] executes the Android applications by a custom application driver and performs a permission usage behavior analysis. CopperDroid [245] traces the system calls and reconstructs the behavior of the target application. TaintDroid [85] and TaintART [244] are taint flow analysis system on different Android Java virtual machines. They track the information flow of the target application at runtime and report the data leakage from sink methods. DexHunter [293] focuses on how to dump the whole DEX file from memory at a "right timing". AppSpear [278] leverages the key data structures in Dalvik to reassemble the DEX file and claims that these data structures are reliable. Both DexHunter and AppSpear assume that there is a clear boundary between the unpacking code and the original code. However, the unpacking code and malicious code may intersperse with each other. Moreover, advanced malware can modify bytecode and data in the DEX file at runtime, and thus the previous dump-based unpacking systems will miss the content modified after the dump procedure.

3.2.3 Hybrid Analysis Tools

Harvester [204] collects runtime values and injects these values into the DEX file for the accuracy improvement of analysis tools. However, some limitations still exist. Firstly, marking logging points and backward slicing are based on the original DEX file. If packing is considered, Harvester loses its target like other static analysis tools. In contrast, DexLego does not analyze the original DEX file. Additionally, Harvester greatly facilitates

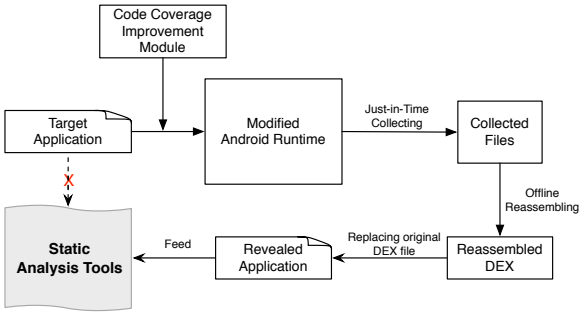


Figure 1: Overview of DexLego.

static analysis tools on solving reflections as they reduce the parameters back into constant strings. However, malware can use advanced reflection code to evade the analysis. Since DexLego replaces the reflective call with direct call, we do not care about how the adversaries use reflection.

3.2.4 Unpacking and Reassembling in Traditional Platforms

Ugarte et al. [256] present a summary of recent unpacking tools and develop an analysis framework for measuring the complexity of a large variety of packers. CoDisasm [50] is a dissembler tool that takes memory snapshot during execution and disassembles the captured memory. Uroboros [264] aims to disassemble binaries with a reassembleable approach. Their reassembling method is based on the disassembling output of Uroboros. DexLego is different from these systems as we do not disassemble the binary or monitor memory. [276] collects the instruction trace at runtime and performs taint analysis on the trace. Unlike [276], DexLego aims to facilitate the other static analysis tools and outputs a standardized DEX file, which could be used for state-of-the-art static analysis tools to perform different kinds of analysis including taint analysis.

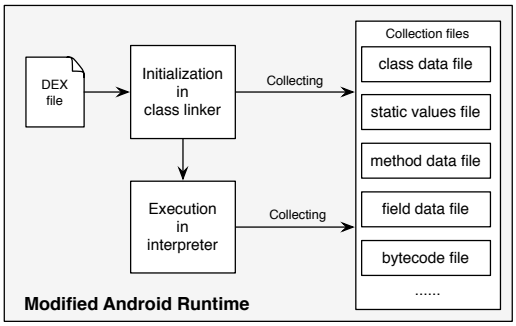


Figure 2: Just-in-Time Collection.

3.3 System Overview

As Figure 1 shows, instead of directly feed the target application to static analysis tools, we firstly execute the target application with DexLego. In executing, we use Just-in-Time (JIT) collection to extract data/instructions and output them to files right before used by ART. In the meantime, we use a code coverage improvement module to increase the code coverage. Next, we reassemble the collected files to a DEX file and use the reassembled DEX file to replace the one in the original APK. Finally, the new APK file is fed to the static analysis tools. The architecture of DexLego contains three main components: 1) the collecting component that collects bytecode and data, 2) the offline reassembling component that reassembles a new DEX file based on the collection result, and 3) the code coverage improvement module that helps DexLego to achieve a high code coverage. Next, we will discuss the three components respectively.

3.3.1 Bytecode and Data Collection

Figure 2 shows the JIT collection we used in DexLego. During the execution of an application, ART firstly extracts the DEX file from the original APK file and passes it to the class linker. The class linker then loads and initializes the classes in the DEX file, and

our JIT collection method collects the metadata of the class (e.g., super class) at this point. Next, when a method is invoked, ART extracts its bytecode from the DEX file, and leverages the interpreter to execute them. The interpreter fetches the entire bytecode (organizing in a 16-bit array) of the method and executes the bytecode instructions one by one. Thus, according to our JIT policy, we collect the executed instructions of the method and their related objects (e.g., `string`) via instruction-level extracting. Note that the execution of the code in the dynamic loaded DEX file also follows the same flow.

The state-of-the-art static analysis tools do not accept machine code as their input. However, ART executes most methods based on the machine code, and the translation from the machine code to the bytecode is a challenging task. To simplify the task, DexLego configures all methods in the application to be executed by the interpreter.

3.3.2 DEX File Reassembling

After the collecting, all the output files are reassembled to a new DEX file offline following the format of a DEX file, and we replace the DEX file in the original APK file with the reassembled one. The modified APK file is finally fed to static analysis tools to study the malicious behavior.

This reassembling is not trivial, and we consider this is the key contribution of this work. In the DEX file format, each method contains only one instruction array. However, due to different control flows (e.g., execution is led to different branches of a branch statement) or self-modifying code, one method may contain different instruction arrays in the collection stage. To correctly combine the collected instructions, we thus design a tree model and a novel collecting and reassembling mechanism. More details are discussed in Section 3.4.1 and Section 3.4.2.

3.3.3 Code Coverage Improvement Module

To improve the code coverage of dynamic analysis systems, there already exists a series of tools or theories like: 1) Input generators or fuzzing tools [6, 41, 97, 105, 157], 2) Symbolic or concolic execution [12, 57, 166, 204, 270, 279] based systems, 3) Force execution [75, 139, 191] based systems. Our code coverage improvement module can be one of them or a combination of them. Note that most of the systems mentioned in 1) and 2) are implemented in Android, and we can directly use them to conduct the execution of the target application with little engineering effort. However, to the best of our knowledge, the idea of force execution has not been applied on Android platform. Thus, we implement a prototype of force execution as a supplement of our code coverage improvement module.

To use force execution in DexLego, we identify the Uncovered Conditional Branches (UCB) and calculate the path to each UCB. By monitoring and manipulating the branch instructions in the interpreter, we force the control flow to go along the calculated path to reach each UCB.

3.4 Design and Implementation

We implement DexLego in an LG Nexus 5X with Android 6.0. Based on the Android Open Source Project [94] (AOSP), we build a customized system image and flash it into the device by leveraging a third-party recovery system [247].

A DEX file consists of data structures that represent different data types used by the interpreter [96]. DexLego collects these data structures directly from memory while they are used by ART at the runtime. Moreover, we leverage instruction-level tracing to collect executed instructions and reassemble them back to a `method` structure. In this section, we

```
1  package com.test;
2
3  public class Main extends Activity {
4    private static final String PHONE = "800-123-456";
5    protected void onCreate(Bundle savedInstanceState) {
6      // ...
7      advancedLeak();
8    }
9
10   public void advancedLeak() {
11     String a = getSensitiveData(); // source
12     for (int i = 0; i < 2; ++i) {
13       normal(a);
14       bytecodeTamper(i);
15     }
16   }
17
18   public void normal(String param) {
19     // do something normal
20   }
21
22   public void sink(String param) {
23     // send param through text message.
24    SmsManager.getDefault().sendTextMessage(PHONE, null, param, null, null); // sink
25   }
26
27   /* While i = 0:
28    *   modify Line 11 to String a = "non-sensitive data"
29    *   modify Line 13 to sink(a)
30    * While i = 1:
31    *   modify Line 11 to String a = getSensitiveData()
32    *   modify Line 13 to normal(a) */
33   public void native bytecodeTamper(int i);
34 }
```

Code 1: An Example of Self-Modifying Code.

discuss 1) bytecode collection, 2) bytecode reassembling, 3) data collection, and 4) DEX file reassembling separately. The approaches to handle reflection and force execution are also discussed in this section.

3.4.1 Bytecode Collection

In ART, after the instruction array of a method is passed to the interpreter, the interpreter executes the instructions one by one following the control flow indicated by them. To expose the behavior of the method, DexLego aims to collect all instructions executed in the method. However, existing systems [278, 293] that use method-level collection cannot defend against dynamic bytecode modification, and the detailed limitation is described as below.

Inadequacy of Method-level Collection. Consider Code 1 as an example. While entering the method `advancedLeak`, the smali code [1] of the method is represented by Code 2. After the first execution of the native method `bytecodeTamper`, the code of the method `advancedLeak` is modified to Code 3. In Code 3, the native method has modified the bytecode to hide the source (Lines 2-4 are changed from Code 2 to Code 3), but the sensitive data is already stored in the register `v0`. During the second execution of the `for` loop, the sensitive data in the register `v0` is leaked through the method `sink` (Lines 9-10 in Code 3). Then, the native method resumes the code back to Code 2. The instruction array of the method `advancedLeak` in memory is either Code 2 or 3 at any time point (e.g., before and after JNI code), which means that the method-level collection (e.g., DexHunter [293] and AppSpear [278]) can only collect Code 2 or 3 even when multiple collections are involved. However, in the static taint flow analysis, the red lines in Code 2 (Lines 2-4) represent a source, but the data fetched from the source are sent to the blue lines (Lines 9-10) which are not a sink. In Code 3, the red lines (Lines 9-10) are a sink, but the received data are obtained from the blue lines (Lines 2-4) which are not a source. Thus, the leak of the sensitive data can be identified from neither Code 2 nor Code 3, and the key reason is that the code representing the source and sink are modified on purpose to hide the taint flow. AppSpear claims that it implements an instruction-level tracing mechanism, however, as we will explain below, simply tracing the instructions does not satisfy the requirement of static analysis tools.

Instruction-level Collection and Tree Model. In light of the shortcoming of method-level collection as described above, the DexLego leverages instruction-level collection to defend

[1] The smali code is a more readable format of the bytecode.

```
.method public advancedLeak()V
  invoke-virtual  p0 , \
    Lcom/test/Main;->getSensitiveData()Ljava/lang/String;
  move-result-object v0
  const/4 v1, 0
  :L0
  const/4 v2, 2
  if-ge v1, v2, :L1
  invoke-virtual  p0, v0 , \
    Lcom/test/Main;->normal(Ljava/lang/String;)V
  invoke-virtual { p0, v1 }, \
    Lcom/ecspride/Main;->bytecodeTamper(I)V
  add-int/lit8 v1, v1, 1
  goto :L0
  :L1
  return-void
.end method
```

Code 2: Smali representation of the method `advancedLeak` while entering and leaving it.

against self-modifying code such as Code 1. One simple approach for instruction-level collection is to list all the executed instructions one by one; however, this approach leads to a code scale issue. Take the loop as an example, since the instructions in a loop are executed for multiple times, the simple approach would lead to a large number of repeating instructions. Moreover, the branch statements and self-modifying code make it possible that different executions of a single method lead to different instruction sequences. However, the format of the DEX file [96] allows only one instruction sequence for a single method.

```
.method public advancedLeak()V
  const-string v0, "non-sensitive data"
  nop
  nop
  const/4 v1, 0
  :L0
  const/4 v2, 2
  if-ge v1, v2, :L1
  invoke-virtual  p0, v0 , \
    Lcom/test/Main;->sink(Ljava/lang/String;)V
  invoke-virtual { p0, v1 }, \
    Lcom/ecspride/Main;->bytecodeTamper(I)V
  add-int/lit8 v1, v1, 1
  goto :L0
  :L1
  return-void
.end method
```

Code 3: Smali representation of the method `advancedLeak` after the first execution of the method `bytecodeTamper`.

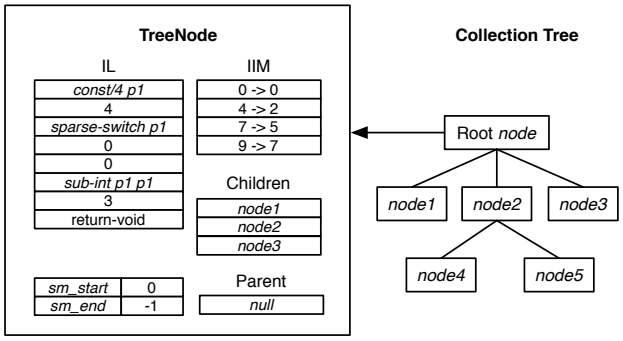


Figure 3: Data Structure Storing All Instructions in a Method During a Single Execution. The right tree structure shows the collection result for a method during a single execution. The left rectangle describes the data structure of each tree node. For each execution of a method, we generate a collection tree.

To address the code scale issue, DexLego eliminates repeating instructions by comparing the instructions with same indices. As mentioned above, the bytecode of a method is organized in a 16-bit unit array and passed to the interpretation functions (`ExecuteSwitchImpl` and `ExecuteGotoImpl` functions). In these functions, the interpreter uses a variable `dex_pc` to represent the index of the executing instruction in the array. In light of this, we identify repeating instructions by comparing the executing instructions with the same `dex_pc` values. Moreover, the self-modifying code can also be identified by the comparison. Different instructions with the same `dex_pc` value actually indicate a runtime modification.

Algorithm 1 illustrates the comparison-based instruction collection algorithm, and Figure 3 shows the related data structures. We consider the first execution of an instruction as a baseline and any different instructions with the same `dex_pc` value as a divergence branch. Thus, each divergence branch indicates a piece of self-modifying code. Note that self-modifying code might also exist in the divergence branch (like multiple layers of self-modifying). The divergence branches in a method then form a tree structure. The right part of Figure 3 shows an example of the final collecting result. Nodes 1-3 represent three

Algorithm 1 Bytecode Collection Algorithm

```
1:  procedure BYTECODECOLLECTION
2:      create node root
3:      current = root
4:      for each executing instruction ins do
5:          let index of ins be dex_pc
6:          if dex_pc exists in current.IIM then
7:              pos_in_IL = current.IIM.get(dex_pc)
8:              old_ins = current.IL.get(pos_in_IL)
9:              if !SameIns(ins, old_ins) then
10:                 create a child node child
11:                 child.parent = current
12:                 child.start_pos = dex_pc
13:                 current = child
14:             else
15:                 continue
16:             end if
17:         else if current has a parent then
18:             parent = current.parent
19:             if dex_pc exists in parent.IIM then
20:                 pos_in_IL = parent.IIM.get(dex_pc)
21:                 old_ins = parent.IL.get(pos_in_IL)
22:                 if SameIns(ins, old_ins) then
23:                     current.end_pos = dex_pc
24:                     current = parent
25:                     continue
26:                 end if
27:             end if
28:         end if
29:         pos_in_IL = current.IL.size()
30:         current.IL.add(ins)
31:         current.IIM.push(pair(dex_pc, pos_in_IL))
32:     end for
33: end procedure
```

pieces of self-modifying code on the root node, and Nodes 4-5 represent two pieces of self-modifying code on Node 2. The left rectangle in Figure 3 shows the `TreeNode` structure which represents a node in the tree structure. The Instruction List (IL) in the structure includes the list of executed instruction and their metadata. The instructions in IL are recorded by the order of their first execution and the IL plays the role of baseline in the node. The `dex_pc` value of an instruction may be different from its index in IL due to branch statements, and we use an Instruction Index Map (IIM) to maintain the mapping

between the instruction's `dex_pc` value and its index in IL for further comparison. `sm_start` and `sm_end` indicate the starting and ending `dex_pc` value of the divergence branch, while `parent` and `children` represent the parent and all children of the node, respectively. With the tree structure, DexLego records all executed instructions in a single execution of a method and maintains the code size similar to the original instruction array.

In Algorithm 1, we only update one node during the execution of a single instruction, and this node is considered as the current node. DexLego creates an empty root node as the current node while entering a method. Once an instruction is executed, we check IIM of the current node to find whether the `dex_pc` value of this instruction has been recorded. If it does not exist in IIM, DexLego pushes the instruction into IL and updates IIM. If the `dex_pc` value already exists in IIM, we add a check procedure to find whether the instruction is the same as the one we recorded before. A positive result means that the same instruction in the same position is executed again, and DexLego does not record it. In contrast, the negative result indicates that modification has occurred to this instruction since its last execution. Then, we create a child node of the current node to represent the divergence branch, and the new node becomes the current node. After that, DexLego treats the instruction as a new instruction and pushes it into IL of the current node. In a divergence branch, another check procedure is added to each instruction, and this check procedure aims to identify whether the current divergence branch converges to its parent. If the same instruction with the same `dex_pc` value has been found in the parent's IL, we consider that the divergence branch converges back to its parent (e.g., current layer of self-modifying code ends) and make the parent node to be the new current node.

Listing 1 shows a high-level semantic view of the collection result of the method

`advancedLeak` in Code 1. When Line 13 in Code 1 is executed for the first time, an invocation of the method `normal` is recorded. Then, in the second run, an invocation of the method `sink` is detected. However, by comparing with the recorded instructions, DexLego finds that it is a divergence point. A child node is forked and the instruction is pushed into the IL of the child node. Furthermore, a convergence point is found when Line 14 is executing. Thus, the collection tree contains a root node and a child node, and the child node contains only one instruction. With the tree, the executed instructions and the control flows in the method are well maintained. Note that the modification to the Line 11 is ignored since the modified instructions are never executed.

```
1 Root Node:
2   String a = getSensitiveData();
3   for (int i = 0; i < 2; ++i) {
4     normal(a);
5     bytecodeTamper(i);
6   }
7
8 Child Node: (Line 13 in Code 1)
9   sink(a);
```

Listing 1: High-level Semantic View of the Collection Result of the Method `advancedLeak` in Code 1.

For the issue of multiple instruction sequences for a single method, we generate multiple collection trees for multiple executions of the method and keep only the unique trees. The trees are further combined together with the approach detailed in Section 3.4.2.

3.4.2 Bytecode Reassembling

The offline reassembling-phase merges the collected trees into a DEX file while holding all the executed instructions and control flows. There are two steps in this phase: 1) converting each tree into an instruction array. 2) merging instruction arrays into the DEX file.

```
String a = getSensitiveData();
for (int i = 0; i < 2; ++i) {
  if (Modification.com_test_Main_advancedLeak_0) {
    normal(a);
  } else {
    sink(a)
  }
  bytecodeTamper(i);
}
```

Code 4: Reassembled Result of the Method `advancedLeak` in Code 1.

Converting a Tree into an Instruction Array. Each node in the collection tree generated from the collection phase contains an independent Instruction List (IL), and the goal of this phase is to combine the ILs in the nodes together without losing any control flows or instructions. To simplify the combination process, we traverse the nodes with the bottom-up fashion since the leaf nodes contain no child node.

To merge a single leaf to its parent, DexLego inserts an additional branch instruction in the divergence point (indicated by `sm_start`, self-modifying start, as defined in the above subsection 3.4.1), with one branch of the instruction pointing to the leaf. To make both conditional branches reachable, the conditional expression of the added branch instruction is calculated based on a static field of an instrument class with random values. Note that the random value produces indeterminacy problem on the additional branch instruction, and we consider it acceptable since the static analysis tool will take both branches of the instruction as reachable.

Once the leaf nodes are recursively merged into their parents, the root node becomes a complete set of the collected instructions including different control flows triggered during the execution.

Code 4 demonstrates the reassembled result of Listing 1. The static field `com_test_Main_advancedLeak_0` in our instrument class `Modification` indicates the divergence point in

Line 13 of Code 1. When this result is fed to static analysis tools, they treat both `normal` and `sink` as reachable and detect the taint flow from sensitive data to text message in Code 1.

Merging Instructions Arrays. For each executed method, the previous phase outputs unique instruction arrays which indicate different executions of the method. Similar to the approach discussed above, we create a method variant for each instruction array and use additional branch instructions to cover different method variants.

3.4.3 Data Collection and DEX Reassembling

As mentioned in Section 3.3.1, besides bytecode instructions, DexLego uses JIT collection to collect the metadata of DEX file. The collected data is written into collection files and further used to reassemble a new DEX file offline.

In Code 1, before any method or field in `Main` is accessed, the class `Lcom/example/Main;` is loaded and initialized. During the process, we firstly store string `Lcom/example/Main;` into a `string` structure and record the index of this `string` structure. Then with the index, a `type` structure is constructed and stored. Finally, a corresponding `class` structure related to the `type` is extracted. The collection occurs again when the class is initialized. The initialization procedure links the methods and fields to the class, and initializes the static fields. In Code 1, methods `onCreate`, `advancedLeak`, `normal`, and `sink` are linked to the class. While the static field `PHONE` is initialized, DexLego stores its name `PHONE`, type `Ljava/lang/String;` and initial value `800-123-456`. Lastly, a `field` structure is created and recorded. The `method` structures and the bytecode inside them are collected before and during the execution of the methods, respectively.

After the collection process, all collection files including bytecode are combined offline

according to the format of the DEX file. Finally, we leverage the Android Asset Packaging Tool integrated with Android SDK to replace the DEX file in the original APK file with the reassembled one. To verify the soundness of our extracting and reassembling algorithm, we perform extensive tests against real-world applications, and the evaluation results in Section 3.5.1, Section 3.5.2, and Section 3.5.4 show that the reassembled DEX file retains the semantics of the real-world application and can be correctly processed by the state-of-the-art static analysis tools.

3.4.4 Handling Reflection

Currently, reflection is a serious obstacle for static analysis tools, and even the state-of-the-art static analysis tools [33, 58, 98, 204] cannot provide a precise result when reflection is involved in an application. FlowDroid [33], DroidSafe [98], and HornDroid [58] can solve the reflection only when the parameters are constant strings. However, the name string can be encrypted in some cases [204], and advanced malware can use reflection without involving any string parameter [83].

The TamiFlex [49] system on traditional Java platform uses load-time instrumentation to log reflective method calls and transform them to direct calls at offline. However, the required load-time instrumentation class `java.lang.instrument` is not supported in Android [33]. Meanwhile, since the target of the reflective method calls is parsed in ART at runtime, DexLego actually knows the target of each reflection. Thus, we apply the similar idea in ART by replacing the reflection calls with direct calls in the collecting stage.

3.4.5 Force Execution

As a supplement of the code coverage improvement module, we implement a prototype of force execution which executes the target application in an iterative fashion. Note that

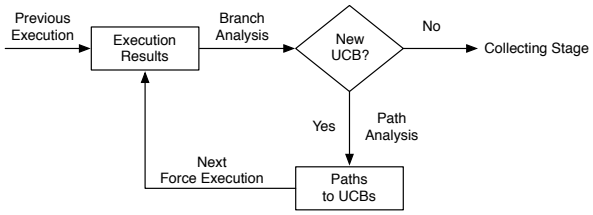


Figure 4: Iterative Force Execution.

our force execution starts from the execution result of the previous execution, and the previous execution could be any kind of execution like fuzzing, symbolic execution, another force execution, or simply open the application and close. Figure 4 shows the workflow of the iterative force execution. In each iteration, we first use branch analysis to identify the Uncovered Conditional Branch (UCB) from the result of the previous execution. Next, we calculate the control flow path to each UCB. A path to an UCB consists of branch instructions and the offsets of the conditional branches leading to the UCB. We save each path into a file and use these files as the input of the next iteration together with the original application. Finally, in the interpretation functions, the outcome of the corresponding conditional expression is automatically manipulated at runtime following the path files. With this approach, DexLego ensures that the runtime control flow goes along the path to the UCB. If no more new UCB are generated after the iteration, we terminate the execution and continue the collecting stage. Otherwise, the next iteration is scheduled.

Since the idea of force execution breaks the normal control flow of the original application, the application may crash due to the control flow falls to an infeasible path [139, 191]. To avoid crash triggered by force execution, we monitor the unhandled exception in the interpreter and tolerate it by directly clear the exception. This strategy helps us to avoid terminations due to infeasible paths while does not affect our runtime bytecode and

data collection.

3.5 Evaluation

In this section, we evaluate DexLego with DroidBench [83] and real-world applications downloaded from Google Play and other application markets. In particular, we aim to answer five research questions:

RQ1. Can we correctly reconstruct the behavior of apps?

RQ3. How is DexLego compared with other tools?

RQ4. Can DexLego work with real-world packed apps?

RQ5. What is the coverage of our force execution prototype?

RQ6. What is the runtime performance overhead?

3.5.1 RQ1: Test with Open-source Apps and Public Packers

To verify the correctness of the reassembled result, we pick up four open source applications (i.e., `HTMLViewer`, `Calculator`, `Calendar`, and `Contacts`) from AOSP [94] and use DexLego to reveal them. By manually comparing the instructions and control flows in each method, we ensure that the instructions and control flows in the source code are completely included in the reassembled result. In regard to `Calendar` and `Contacts`, we use Soot framework [143] to build a complete call graph since the numbers of instructions (78,598 and 103,602 instructions, respectively) are too large for a manual analysis. By examining the call graph, we confirm that the control flows in these two applications are properly maintained in the reassembled DEX.

Next, to check the functionality against packers, we use different public packing platforms to pack these applications and then use DexLego to reveal them again. Table 1

Table 1: Test Result of Different Packers.

Applications	HTMLViewer	Calculator	Calendar	Contacts
# of Instructions	217	2,507	78,598	103,602
360 [198]	✓	✓	✓	✓
Alibaba [5]	✓	✓	✓	✓
Tencent [248]	✓	✓	✓	✓
Baidu [42]	✓	✓	✓	✓
Bangcle [44]	✓	✓	✓	✓
NetQin [179]	The service is offline now			
APKProtect [14]	Unresponsive to packing requests			
Ijiami [114]	Samples are rejected by human agents			

shows the result of the experiments. For the packers including 360 [198], Alibaba [5], Tencent [248], Baidu [42], and Bangcle [44], DexLego succeeds in both collection and reassembling stages. By using the same approach described above, we ensure that DexLego correctly rebuilds the behavior of each application. Note that NetQin packer [179] mentioned in AppSpear [278] is no longer available. The APKProtect [14] is unresponsive to the packing requests, and there are no logs of the occurred errors. The packing service provided by Ijiami [114] requires manual audits by their agents, and they reject our applications for the reason that the applications are not actually developed by us.

3.5.2 RQ2: Test with Existing Tools

Static Analysis Tools

DroidBench [83] is a set of open-source samples that leak sensitive data in various ways. It is considered as a benchmark for Android application analysis and widely used among recent analysis tools [33, 58, 98, 148, 265]. The latest release of DroidBench contains 119 applications, including both leaky and benign samples. The leaky samples leak a variety of sensitive data fetched from sources (API calls that fetch sensitive information)

Table 2: Analysis Result of Static Analysis Tools. The columns in "Original" represent the analysis result of the original samples, and the columns in "DexLego" represent that of the samples reassembled by DexLego. The column "TP" and "FP" indicate the number of true positives and false positives of the analysis result, respectively.

	# of Samples	# of Malware	Original		DexLego	
			TP	FP	TP	FP
FlowDroid [33]	134	111	81	10	95	4
DroidSafe [98]	134	111	95	12	105	7
HornDroid [58]	134	111	98	9	106	4

to sinks (API calls that may leak information), and the benign samples contain no such information flows. As a supplement, we contribute another 15 samples involving usage of advanced reflection (5 samples), dynamic loading (3 samples), self-modifying (4 samples), and unreachable taint flows (3 samples). Current static analysis tools [33, 58, 98] cannot precisely analyze these newly added samples. Besides this benchmark, we choose three representative static analysis tools (FlowDroid [33], DroidSafe [98], and HornDroid [58]) to conduct the experiments.

Our experiment involves 134 samples (119 samples in the newest release plus 15 samples we contributed) in DroidBench. Since the lines of code in DroidBench samples are small, we simply choose the state-of-the-art fuzzing tool Sapienz [158] to generate the inputs for the execution. We first use the static analysis tools to analysis the original samples and the samples processed by DexLego, and the result is shown in Table 2. The table shows that DexLego increases more than 8 true positives by resolving advanced reflections, extracting self-modifying code and dynamic loading code. Moreover, The JIT collection ensures that the extracted data reflects the performed behavior of the target application. Thus, at least 5 false positives introduced by dead code blocks are removed. Next, with-

out losing generality, we use one of the most popular packers tested in Section 3.5.1, 360 packer, to pack the original samples and process the packed samples with DexLego, DexHunter [293], and AppSpear [278], respectively. The analysis result of the processed samples is shown in Table 3. Note that DexHunter and AppSpear lead to the same result since they can extract the original DEX files and the result is same as analyzing the original DEX. Compared to DexLego, they fail to deal with self-modifying code and reflection. As shown in the table, DexLego provides more than 5 true positives and reduces more than 5 false positives than DexHunter and AppSpear. We note that DexLego fails to cover taint flow in only one application among all samples. In this sample, sensitive data only leaks in the tablet, and it cannot be detected as we execute it in a mobile phone.

$$\begin{aligned} Sensitivity = \frac{tp}{tp + fn},\ Specificity = \frac{tn}{tn + fp}, \\ \textit{F-Measure} = 2 \times \frac{Sensitivity \times Specificity}{Sensitivity + Specificity} \end{aligned} \tag{3.1}$$

The F-Measure [58] is a standard measure of the performance of a classification, and it is calculated by Formula (3.1). Figure 5 illustrates the changes of F-Measures after involving DexHunter, AppSpear, and DexLego. Once DexLego is involved, the F-Measure of FlowDroid increases from 63% to 84% on DroidBench, and that of DroidSafe increases from 61% to 80%. In regard to the most recent static analysis tool, HornDroid, the F-Measure increases from 72% to 89%. The percentages of incremental values are 33.3%, 31.1%, and 23.6%, respectively. In the meantime, the improvement introduced by DexHunter and AppSpear is less than 3%.

Table 3: Analysis Result of Packed Samples. The columns in "DH", "AS", and "DexLego" represent the analysis result of the samples processed by DexHunter [293], AppSpear [278], and DexLego, respectively. The column "TP" and "FP" indicate the number of true positives and false positives of the analysis result, respectively.

	# of Samples	# of Malware	DH [293] / AS [278]		DexLego	
			TP	FP	TP	FP
FlowDroid [33]	134	111	84	10	95	4
DroidSafe [98]	134	111	98	12	105	7
HornDroid [58]	134	111	101	9	106	4

Dynamic Analysis Tools

Dynamic analysis tools can be circumvented through implicit taint flows, and a recent work [204] shows that a representitive dynamic analysis tool, TaintDroid [85], misses leakage on some samples of DroidBench. We pick these samples and analyze them with both TaintDroid and another recent dynamic analysis tool TaintART [244]. Next, we use DexLego to analyze it again. The reassembled result is fed to HornDroid, the most recent static analysis tool, for comparison.

Table 4 shows the taint flow analysis results of TaintDorid, TaintART, and combing DexLego and HornDroid. As shown in Table 4, the static analysis result of reassembled APK file by DexLego detects the taint flows and is more precise than dynamic analysis tools. In `Button1` and `Button3`, the sensitive data are leaked via callback methods, and we solve it properly while the dynamic analysis tools miss it. As TaintDroid executes applications on emulator, the sample `EmulatorDetection1` evades the analysis. Both TaintDroid and TaintART cannot detect the implicit taint flows in `ImplicitFlow1`, and using HornDroid with DexLego provides a precise analysis result. One of the taint flows in `PrivateDataLeak3` leaks the sensitive data through writing/reading an external file, and all tested tools fail to

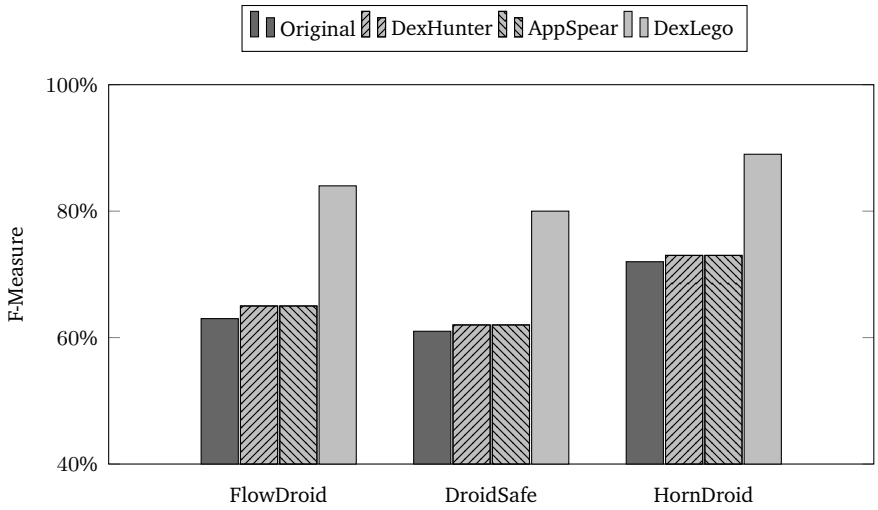


Figure 5: F-measures of Static Analysis Tools.

Table 4: Analysis Result of Dynamic Analysis Tools and DexLego. The columns "TD" and "TA" represent the taint flows detected by TaintDroid [85] and TaintART [244], respectively. The last column shows the detected taint flows by feeding the revealed result of DexLego to HornDroid [58].

Samples	Leak #	# of Leak Detected		
		TD [85]	TA [244]	DexLego + HD [58]
Button1	1	0	0	1
Button3	2	0	0	2
EmulatorDetection1	1	0	1	1
ImplicitFlow1	2	0	0	2
PrivateDataLeak3	2	1	1	1

detect this flow since they do not take this case into account. Note that these missed taint flows are not caused by code coverage issue, but due to the weakness of dynamic analysis tools on implicit taint flows.

Note that DexLego is not a dynamic analysis tool. We believe we should not directly compare DexLego with dynamic analysis tools, and the dynamic analysis tools have their advantages. However, the experiment conducted in this subsection is to show that DexLego can help static analysis tools make up some deficiencies of dynamic analysis tools.

3.5.3 RQ3: Test with Real-world Packed Applications

A previous work [259] has downloaded more than one million applications from Google Play by a crawler in 2014, and we select the packed applications from this set. Since the DEX file in an application packed by the public packing platforms contains only the classes needed to unpack the original DEX file, the number of the classes in the DEX file is less compared to normal applications. In light of this, we perform a coarse-grain analysis to screen the applications which contains less than 50 classes from the top rated $10,000$ applications. Next, we select the first 9 applications from the screened result by manually checking and reverse engineering. Without loss of generality, we download the latest version of these applications from three different popular application markets: 1) Google Play [93] (denoted as set `A`), 2) 360 Application Market [199] (denoted as set `B`), and 3) Wandoujia Application Market [261] (denoted as set `C`).

For these real-world packed applications, we use FlowDroid to provide a quick scan on the original applications, and then execute them with DexLego for 5 minutes. Next, the reassembled APK file is analyzed again by FlowDroid. Table 5 shows the result of our experiment. Although no taint flow can be detected from the original samples, FlowDroid detects several taint flows from these revealed applications. From the analysis result, we find that all of these applications send device ID (IMEI number) to remote servers. Moreover, three of them leak location information and two of them leak SSID. This result also shows that DexLego successfully reveals the latest packed real-world applications.

Table 5: Analysis Result of Packed Real-world Applications. The column "Sample Set" is defined in Section 3.5.3, which indicates the source of the application. The column "# of Installs" shows the installation number provided by the application markets. The column "Original" represents the number of detected taint flows in the original application while the column "Revealed" is the number of detected taint flows in the revealed APK file.

Package Name	Version	Sample Set	# of Installs	Original	Revealed
com.lenovo.anyshare	3.6.68	A	100 million	0	4
com.moji.mjweather	6.0102.02	A	1 million	0	5
com.rongcai.show	3.4.9	A	100 thousand	0	3
com.wawoo.snipershootwar	2.6	B	10 million	0	4
com.wawoo.gunshootwar	2.6	B	10 million	0	5
com.alex.lookwifipassword	2.9.6	B	100 thousand	0	2
com.gome.eshopnew	4.3.5	C	15.63 million	0	3
com.szzc.ucar.pilot	3.4.0	C	3.59 million	0	5
com.pingan.pabank.activity	2.6.9	C	7.9 million	0	14

3.5.4 RQ4: Code Coverage

To evaluate the code coverage of our force execution engine, we pick up five open source applications from the random page [87] of F-Droid [86] project. For each application, we first execute it with Sapienz [158] and use Java Code Coverage Library (JaCoCo) [125] for Android Studio to calculate the coverage. Next, based on the result of Sapienz, we execute it again using the force execution engine as the code coverage improvement module.

Table 6 shows the details of the samples including package name, version number, the number of instructions, and the total size of the dump files after fuzzing by Sapienz. Note that the size of the dump files is not only related to the number of the instructions in the application, but also related to the size of other data structures in the DEX file (e.g., number of classes, number of methods, size of strings, and so on.) and the code coverage of the fuzzing. Table 7 shows the average coverage of these samples with different granularities.

Table 6: Samples from F-Droid [86].

Package Name	Version	# of Instructions	Dump File Size
be.ppareit.swiftp	2.14.2	8,812	47.26 KB
fr.gaulupeau.apps.InThePoche	2.0.0b1	29,231	771.81 KB
org.gnucash.android	2.1.7	56,565	2.40 MB
org.liberty.android.fantastischmemopro	10.9.993	57,575	1.55 MB
com.fastaccess.github	2.1.0	93,913	3.18 MB

Table 7: Code Coverage with F-Droid Applications.

	Class	Method	Line	Branch	Instruction
Sapienz [158]	44%	37%	32%	20%	32%
Sapienz + DexLego	87%	88%	82%	78%	82%

The results show that the force execution significantly improves the coverage and achieves an average instruction coverage of 82%. By manually check the source code, we group the cause of missed instructions into three main categories: 1) Dead code blocks. As an example, the `CmdTemplate` class is never involved in the application `be.ppareit.swiftp`, thus the entire instructions in this class are not included while calculating coverage. 2) Native crashes. Although DexLego clears the unhandled exceptions in the interpreter, the abnormal control flows may lead the native code to crash. This may be mitigated by the on demand runtime memory allocation mechanism applied in [191]. 3) Instructions in exception handlers. During force execution, the expected exceptions in the `try-catch` blocks may not be thrown due to abnormal control flow, and it may be solved by treating these blocks as branch instructions in the branch analysis. We leave it as a future work.

3.5.5 RQ5: Performance

As DexLego traces and extracts instructions at runtime, it slows the ART during instruction execution. To learn the performance overhead introduced by DexLego, we use

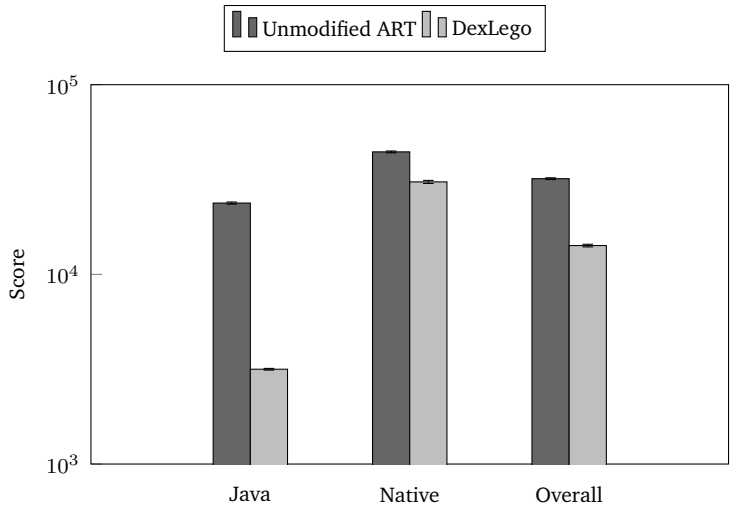


Figure 6: Performance Measured by CF-Bench [61].

Table 8: Time Consumption of DexLego. The column "Original" represents the mean and standard deviation (STD) of the launch time with unmodified ART, while the last column represents launch time with DexLego.

Application	Version	Original		With DexLego	
		Mean	STD	Mean	STD
Snapchat	9.43.0.0	826.9ms	52.11ms	1,664.7ms	16.08ms
Instagram	9.7.0	608.5ms	45.6ms	1,275.8ms	25.37ms
WhatsApp	2.16.310	236.4ms	12.24ms	480.2ms	84.3ms

CF-Bench [61] to compare the performance of the unmodified ART and ART with DexLego. For each environment, we run CF-Bench for 30 times, and the results are presented in Figure 6. A higher score indicates a better performance. It shows that DexLego brings 7.5x, 1.4x, 2.3x overhead on Java score, native score, and overall score, respectively.

Moreover, we evaluate the launch time of three popular applications (i.e., Snapchat, Instagram, and WhatsApp) downloaded from Google Play. While an activity in an application is launching, the `ActivityManager` reports the time usage for initializing and displaying. We launch each application for 30 times and the result is summarized in Table 8. The result shows that DexLego introduces about two times slowdown on the launch time, and this

result matches the overall overhead tested by CF-Bench.

Since our system is designed for security analyst instead of traditional users, we do not take performance as a critical factor. In summary, we consider the overhead is acceptable and leave the further improvement as our future work.

CHAPTER 4 TRANSPARENT MALWARE ANALYSIS ON ARM

4.1 Introduction

We consider that an analysis system consists of an *Environment* (e.g., operating system, emulator, hypervisor, or sandbox) and an *Analyzer* (e.g., instruction analyzer, API tracer, or application debugger). The *Environment* provides the *Analyzer* with the access to the states of the target malware, and the *Analyzer* is responsible for the further analysis of the states. Consider an analysis system that leverages the emulator to record the system call sequence and sends the sequence to a remote server for further analysis. In this system, the *Environment* is the emulator, which provides access to the system call sequence, and both the system call recorder and the remote server belong to the *Analyzer*. Evasive malware can detect this analysis system via anti-emulation techniques and evade the analysis.

To build a transparent analysis system, we propose three requirements. Firstly, the *Environment* must be isolated. Otherwise, the *Environment* itself can be manipulated by the malware. Secondly, the *Environment* exists on an off-the-shelf (OTS) bare-metal platform without modifying the software or hardware (e.g., emulation and virtualization are not). Although studying the anti-emulation and anti-virtualization techniques [131, 193, 224, 258] helps us to build a more transparent system by fixing the imperfections of the *Environment*, we consider perfect emulation or virtualization is impractical due to the complexity of the software. Instead, if the *Environment* already exists in the OTS bare-metal platform, malware cannot detect the analysis system by the presence of the *Environment*. Finally, the *Analyzer* should not leave any detectable footprints (e.g., files, memory, registers, or code) to the outside of the *Environment*. An *Analyzer* violating this requirement can be detected.

In light of the three requirements, we present Ninja [2], a transparent malware analysis framework on ARM platform based on hardware features including TrustZone technology, Performance Monitoring Unit (PMU), and Embedded Trace Macrocell (ETM). We implement a prototype of Ninja that embodies a trace subsystem with different tracing granularities and a debug subsystem with a GDB-like debugging protocol on ARM Juno development board. Additionally, hardware-based traps and memory protection are leveraged to keep the use of system registers transparent to the target application. The experimental results show that our framework can transparently monitor and analyze the behavior of the malware samples. Moreover, Ninja introduces reasonable overhead. We evaluate the performance of the trace subsystem with several popular benchmarks, and the result shows that the overheads of the instruction trace and system call trace are less than 1% and the Android API trace introduces 4 to 154 times slowdown.

4.2 Related Work

4.2.1 Transparent Malware Analysis on x86

Ether [77] leverages hardware virtualization to build a malware analysis system and achieves high transparency. Spider [76] is also based on hardware virtualization, and it focuses on both applicability and transparency while using memory page instrument to gain higher efficiency. Since the hardware virtualization has transparency issues, these systems are naturally not transparent. LO-PHI [236] leverages additional hardware sensors to monitor the disk operation and periodically poll memory snapshots, and it achieves a higher transparency at the cost of incomplete view of system states.

[2] A Ninja in feudal Japan has invisibility and transparency ability

MalT [285] increases the transparency by involving System Manage Mode (SMM), a special CPU mode in x86 architecture. It leverages PMU to monitor the program execution and switch into SMM for analysis. Comparing with MalT, Ninja improves in the following aspects: 1) The PMU registers on MalT are accessible by privileged malware, which breaks the transparency by checking the values of these registers. By leveraging TrustZone technology, Ninja configures needed PMU registers as secure ones so that even the privileged malware in the normal domain cannot access them. 2) MalT is built on SMM. However, SMM is not designed for security purpose such as transparent debugging (originally for power management); frequent CPU mode switching introduces a high performance overhead (12 μs is required for a SMM switch [285]). Ninja is based on TrustZone, a dedicated security extension on ARM. The domain switching only needs 0.34 μs. 3) Besides a debugging system, Ninja develops a transparent tracing system with existing hardware. The instruction and system call tracing introduce negligible overhead, which is immune to timing attacks while MalT suffers from external timing attack.

BareCloud [141] and MalGene [140] focus on detecting evasive malware by executing malware in different environments and comparing their behavior. There are limitations to this approach. Firstly, it fails to transparently fetch the malware runtime behavior (e.g., system calls and modifications to memory/registers) on a bare-metal environment. Secondly, it assumes that the evasive malware shows the malicious behavior in at least one of the analysis platforms. However, sophisticated malware may be able to detect all the analysis platforms and refuse to exhibit any malicious behavior during the analysis. Lastly, after these tools identify the evasive malware from the large-scale malware samples, they still need a transparent malware analysis tool which is able to analyze these evasive sam-

ples transparently. Ninja provides a transparent framework to study the evasive malware and plays a complementary role for these systems.

4.2.2 Dynamic Analysis Tools on ARM

Emulation-based systems. DroidScope [277] rebuilds the semantic information of both the Android OS and the Dalvik virtual machine based on QEMU. CopperDroid [245] is a VMI-based analysis tool that automatically reconstructs the behavior of Android malware including inter-process communication (IPC) and remote procedure call interaction. DroidScibe [73] uses CopperDroid [245] to collect behavior profiles of Android malware, and automatically classifies them into different families. Since the emulator leaves footprints, these systems are natural not transparent.

Hardware virtualization. Xen on ARM [273] migrates the hardware virtualization based hypervisor Xen to ARM architecture and makes the analysis based on hardware virtualization feasible on mobile devices. KVM/ARM [72] uses standard Linux components to improve the performance of the hypervisor. Although the hardware virtualization based solution is considered to be more transparent than the emulation or traditional virtualization based solution, it still leaves some detectable footprints on CPU semantics while executing specific instructions [224].

Bare-metal systems. TaintDroid [85] is a system-wide information flow tracking tool. It provides variable-level, message-level, method-level, and file-level taint propagation by modifying the original Android framework. TaintART [244] extends the idea of TaintDroid on the most recent Android Java virtual machine Android Runtime (ART). VetDroid [294] reconstructs the malicious behavior of the malware based on permission usage, and it is applicable to taint analysis. DroidTrace [295] uses `ptrace` to monitor the dynamic load-

ing code on both Java and native code level. BareDroid [169] provides a quick restore mechanism that makes the bare-metal analysis of Android applications feasible at scale. Although these tools attempt to analyze the target on real-world devices to improve transparency, the modification to the Android framework leaves some memory footprints or code signatures, and the ptrace-based approaches can be detected by simply check the `/proc/self/status` file. Moreover, these systems are vulnerable to privileged malware.

4.2.3 TrustZone-related Systems

TZ-RKP [38] runs in the secure domain and protects the rich OS kernel by event-driven monitoring. Sprobes [275] provides an instrumentation mechanism to introspect the rich OS from the secure domain, and guarantees the kernel code integrity. SeCReT [129] is a framework that enables a secure communication channel between the normal domain and the secure domain, and provides a trust execution environment. Brasser *et al*. [52] use TrustZone to analyze and regulate guest devices in a restricted host spaces via remote memory operation to avoid misusage of sensors and peripherals. C-FLAT [4] fights against control-flow hijacking via runtime control-flow verification in TrustZone. TrustShadow [103] shields the execution of an unmodified application from a compromised operating system by building a lightweight runtime system in the ARM TrustZone secure world. The runtime system forwards the requests of system services to the commodity operating systems in the normal world and verifies the returns. Unlike previous systems, Ninja leverage TrustZone to transparently debug and analyze the ARM applications and malware.

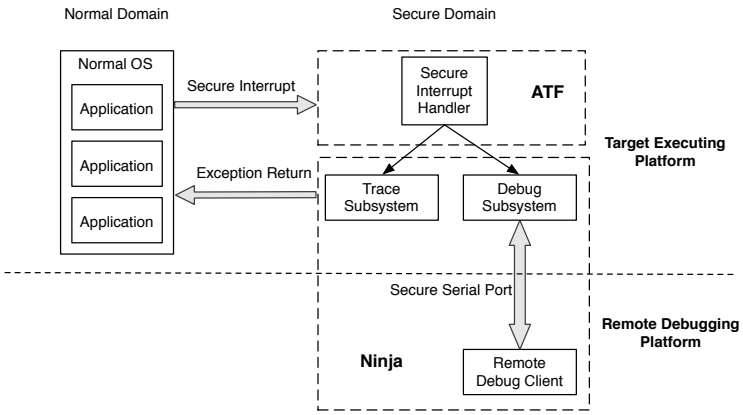


Figure 7: Architecture of Ninja.

4.3 System Architecture

Figure 7 shows the architecture of Ninja. The Ninja consists of a target executing platform and a remote debugging client. In the target executing platform, TrustZone provides hardware-based isolation between the normal and secure domains while the rich OS (e.g., Linux or Android) runs in the normal domain and Ninja runs in the secure domain. We setup a customized exception handler in EL3 to handle asynchronous exceptions (i.e., interrupts) of our interest. Ninja contains a Trace Subsystem (TS) and a Debug Subsystem (DS). The TS is designed to transparently trace the execution of a target application, which does not need any human interaction during the tracing. This feature is essential for automatic large-scale analysis. In contrast, the DS relies on human analysts. In the remote debugging platform, the analysts send debug commands via a secure serial port and the DS then response to the commands. During the execution of an application, we use secure interrupts to switch into the secure domain and then resume to the normal domain by executing the exception return instruction `eret`.

4.3.1 Reliable Domain Switch

Normally, the `smc` instruction is used to trigger a domain switch by signaling a Secure Monitor Call (SMC) exception which is handled in EL3. However, as the execution of the `smc` instruction may be blocked by privileged malware, this software-based switch is not reliable.

Another solution is to trigger a secure interrupt which is considered as an asynchronous exception in EL3. ARM Generic Interrupt Controller (GIC) [27] partitions all interrupts into secure group and non-secure group, and each interrupt is configured to be either secure or non-secure. Moreover, the GIC Security Extensions ensures that the normal domain cannot access the configuration of a secure interrupt. Regarding to Ninja, we configure PMI to be a secure interrupt so that an overflow of the PMU registers leads to a switch to the secure domain. To increase the flexibility, we also use similar technology mentioned in [241] to configure the General Purpose Input/Output (GPIO) buttons as the source of secure Non-Maskable Interrupt (NMI) to trigger the switch. The switch from secure domain to normal domain is achieved by executing the exception return instruction `eret`.

4.3.2 The Trace Subsystem

The Trace Subsystem (TS) provides the analyst the ability to trace the execution of the target application in different granularities during automatic analysis including instruction tracing, system call tracing, and Android API tracing. We achieve the instruction and system call tracing via hardware component ETM, and the Android API tracing with help of PMU registers.

By default, we use the GPIO button as the trigger of secure NMIs. Once the button is pressed, a secure NMI request is signaled to the GIC, and GIC routes this NMI to EL3. Ninja toggles the enable status of ETM after receiving this interrupt and outputs the tracing result if needed. Additionally, the PMU registers are involved during the Android API trace. Note that the NMI of GPIO buttons can be replaced by any system events that trigger an interrupt (e.g., system calls, network events, clock events, and etc.), and these events can be used to indicate the start or end of the trace in different usage scenarios.

Another advanced feature of ETM is that PMU events can also be configured as an external input source. In light of this, we specify different granularities of the tracing. For example, we trace all the system calls by configure the ETM to use the signal of PMU event `EXC_SVC` as the external input.

4.3.3 The Debug Subsystem

In contrast to the TS, the Debug Subsystem (DS) is designed for manual analysis. It establishes a secure channel between the target executing platform and the remote debugging platform, and provides a user interface for human analysts to introspect the execution status of the target application.

To interrupt the execution of the target, we configure the PMI to be secure and adjust the value of the PMU counter registers to trigger an overflow at a desired point. Ninja receives the secure interrupt after a PMU counter overflows and pauses the execution of the target. A human analyst then issues debugging commands via the secure serial port and introspects the current status of the target following our GDB-like debugging protocol. To ensure the PMI will be triggered again, the DS sets desirable values to the PMU registers before exiting the secure domain.

Moreover, similar to the TS, we specify the granularity of the debugging by monitoring different PMU events. For example, if we choose the event `INST_RETIRED` which occurs after an instruction is retired, the execution of the target application is paused after each instruction is executed. If the event `EXC_SVC` is chosen, the DS takes control of the system after each system call.

4.4 Design and Implementation

We implement Ninja on a 64-bit ARMv8 Juno r1 board. There are two ARM Cortex-A57 cores and four ARM Cortex-A53 cores on the board, and all of them include the support for PMU, ETM, and TrustZone. Based on the ATF and Linaro's deliverables on Android 5.1.1 for Juno, we build a customized firmware for the board. Note that Ninja is compatible with commercial mobile devices because it relies on existing deployed hardware features.

4.4.1 Bridge the Semantic Gap

As with the VMI-based [126] and TEE-based [285] systems, bridging the semantic gap is an essential step for Ninja to conduct the analysis. In particular, we face two layers of semantic gaps in our system.

Gap between Normal and Secure Domains

In the DS, Ninja uses PMI to trigger a trap to EL3. However, the PMU counts the instructions executed in the CPU disregarding to the current running process. That means the instruction which triggers the PMI may belong to another application. Thus, we first need to identify if the current running process is the target. Since Ninja is implemented in the secure domain, it cannot understand the semantic information of the normal domain, and we have to fill the semantic gap to learn the current running process in the OS.

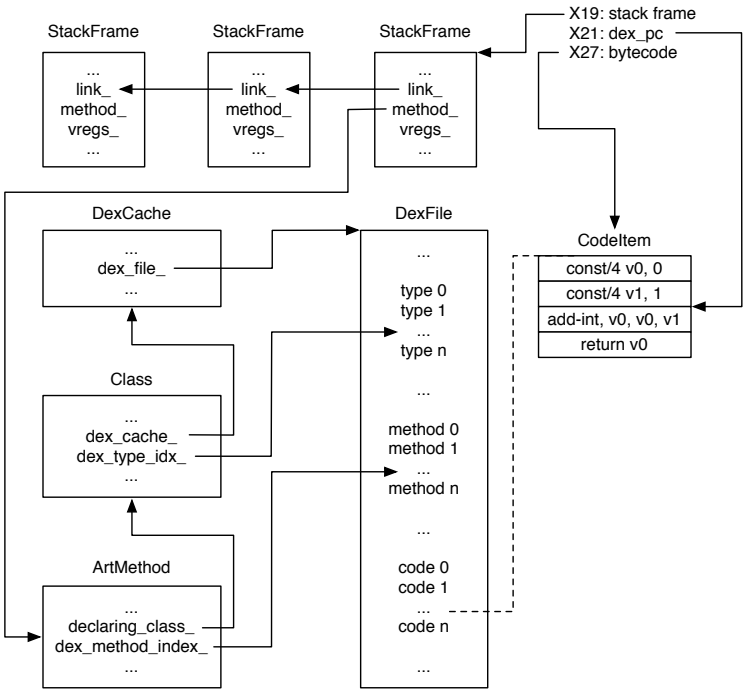


Figure 8: Semantics in the Function `ExecuteGotoImpl`.

In Linux, each process is represented by an instance of `thread_info` data structure, and the one for the current running process could be obtained by `SP & ~(THREAD_SIZE - 1)`, where `SP` indicates the current stack pointer and `THREAD_SIZE` represents the size of the stack. Next, we can fetch the `task_struct`, which maintains the process information (like pid, name, and memory layout), from the `thread_info`. Then, the target process can be identified by the pid or process name.

Gap in Android Java Virtual Machine

Android maintains a Java virtual machine to interpret Java bytecode, and we need to figure out the current executing Java method and bytecode during the Android API tracing and bytecode stepping. DroidScope [277] fills the semantic gaps in the Dalvik to understand the current status of the VM. However, as a result of Android upgrades, Dalvik is no longer available in recent Android versions, and the approach in DroidScope is not

applicable for us.

By manually analyzing the source code of ART, we learn that the bytecode interpreter uses `ExecuteGotoImpl` or `ExecuteSwitchImpl` function to execute the bytecode. The approaches we used to fill the semantic gap in these two functions are similar, and we use function `ExecuteGotoImpl` as an example to explain our approach. In Android, the bytecode of a Java method is organized as a 16-bit array, and ART passes the bytecode array to the function `ExecuteGotoImpl` together with the current execution status such as the current thread, caller and callee methods, and the call frame stack that stores the call stack and parameters. Then, the function `ExecuteGotoImpl` interprets the bytecode in the array following the control flows, and a local variable `dex_pc` indicates the index of the current interpreting bytecode in the array. By manual checking the decompiled result of the function, we find that the pointer to the bytecode array is stored in register `X27` while variable `dex_pc` is kept by register `X21`, and the call frame stack is maintained in register `X19`. Figure 8 shows the semantics in the function `ExecuteGotoImpl`. By combining registers `X21` and `X27`, we can locate the current executing bytecode. Moreover, a single frame in the call frame stack is represented by an instance of `StackFrame` with the variable `link_` pointing to the previous frame. The variable `method_` indicates the current executing Java method, which is represented by an instance of `ArtMethod`. Next, we fetch the declaring class of the Java method following the pointer `declaring_class_`. The pointer `dex_cache_` in the declaring class points to an instance of `DexCache` which is used to maintain a cache for the DEX file, and the variable `dex_file_` in the `DexCache` finally points to the instance of `DexFile`, which contains all information of a DEX file. Detail description like the name of the method can be fetched via the index of the method (i.e., `dex_method_index_`)

in the method array maintained by the `DexFile`. Note that both `ExecuteGotoImpl` and `ExecuteSwitchImpl` functions have four different template implementations in ART, and our approach is applicable to all of them.

4.4.2 Secure Interrupts

In GIC, each interrupt is assigned to Group 0 (secure interrupts) or Group 1 (non-secure interrupts) by a group of 32-bit `GICD_IGROUPR` registers. Each bit in each `GICD_IGROUPR` register represents the group information of a single interrupt, and value 0 indicates Group 0 while value 1 means Group 1. For a given interrupt ID `n`, the index of the corresponding `GICD_IGROUPR` register is given by $n \,/\, 32$, and the corresponding bit in the register is $n \bmod 32$. Moreover, the GIC maintains a target process list in `GICD_ITARGETSR` registers for each interrupt. By default, the ATF configures the secure interrupts to be handled in Cortex-A57 core 0.

As mentioned in Section 4.3.1, Ninja uses secure PMI and NMI to trigger a reliable switch. As the secure interrupts are handled in Cortex-A57 core 0, we run the target application on the same core to reduce the overhead of the communication between cores. In Juno board, the interrupt ID for PMI in Cortex-A57 core 0 is 34. Thus, we clear the bit 2 of the register `GICD_IGROUPR1` ($34 \bmod 32 = 2, 34 \,/\, 32 = 1$) to mark the interrupt 34 as secure. Similarly, we configure the interrupt 195, which is triggered by pressing a GPIO button, to be secure by clearing the bit 3 of the register `GICD_IGROUPR6`.

4.4.3 The Trace Subsystem

Instruction Tracing

Ninja uses ETM embedded in the CPU to trace the executed instructions. Figure 9

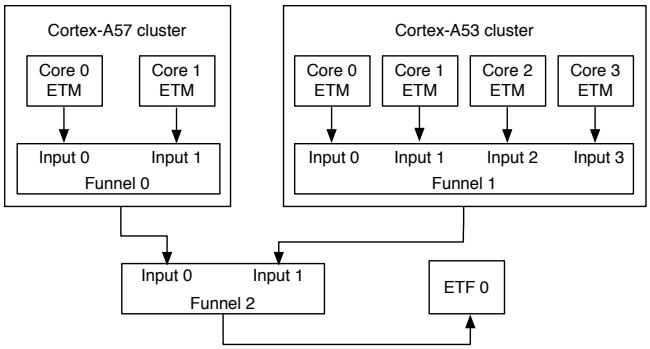


Figure 9: ETM in Juno Board.

shows the ETM and related components in Juno board. The funnels shown in the figure are used to filter the output of ETM, and each of them is controlled by a group of CoreSight Trace Funnel (CSTF) registers [22]. The filtered result is then output to Embedded Trace FIFO (ETF) which is controlled by Trace Memory Controller (TMC) registers [23].

In our case, as we only need the trace result from the core 0 in the Cortex-A57 cluster, we set the `EnS0` bit in CSTF Control Register of funnel 0 and funnel 2, and clear other slave bits. To enable the ETF, we set the `TraceCaptEn` bit of the TMC CTL register.

The ETM is controlled by a group of trace registers. As the target application is always executed in non-secure EL0 or non-secure EL1, we make the ETM only trace these states by setting all `EXLEVEL_S` bits and clearing all `EXLEVEL_NS` bits of the `TRCVICTLR` register. Then, Ninja sets the `EN` bit of `TRCPRGCTLR` register to start the instruction trace. In regard to stop the trace, we first clear the `EN` bit of `TRCPRGCTLR` register to disable ETM and then set the `StopOnFl` bit and the `FlushMan` bits of `FFCR` register in the TMC registers to stop the ETF. To read the trace result, we keep reading from `RRD` register until `0xFFFFFFFF` is fetched. Note that the trace result is an encoded trace stream, and we use an open source analyzer `ptm2human` [112] to convert the stream to a readable format.

System Call Tracing

The system call of Linux in ARM platforms is achieved by supervisor call instruction `svc`, and an immediate value following the `svc` instruction indicates the corresponding system call number. Since the ETM can be configured to trace the PMU event `EXC_SVC`, which occurs right after the execution of a `svc` instruction, we trace the system calls via tracing this event in ETM.

As mentioned in Section 4.3.2, we can configure the ETM to trace PMU events during the instruction trace. The `TRCEXTINSELR` register is used to trace at most four external input source, and we configure one of them to trace the `EXC_SVC` event. In Cortex-A57, the event number of the `EXC_SVC` event is `0x60`, so we set the `SEL0` bits of the `TRCEXTINSELR` register to be `0x60`. Also, the `SELECT` bits of the second trace resource selection control register `TRCRSCTLR2` (`TRCRSCTLR0` and `TRCRSCTLR1` are reserved) is configured to 0 to select the external input 0 as tracing resource 2. Next, we configure the `EVENT0` bit of `TRCEVENTCTL0R` register to 2 to select the resource 2 as event 0. Finally, the `INSTEN` bit of `TRCEVENTCTL1R` register is set to 0x1 to enable event 0. Note that the `X` bit of PMU register `PMCR_EL0` should also be set to export the events to ETM. After the configuration, the ETM can be used to trace system calls, and the configuration to start and stop the trace is similar to the one in instruction tracing.

Android API Tracing

Unlike the instruction trace and system call trace, we cannot use ETM to directly trace the Android APIs as the existence of the semantic gap. As mentioned in Section 4.4.1, each Java method is interpreter by `ExecuteGotoImpl` or `ExecuteSwitchImpl` function, and ART

jumps to these functions by a branch instruction `bl`. Since a PMU event `BR_RETIRED` is fired after execution of a branch instruction, we use PMU to trace the `BR_RETIRED` event and reconstruct the semantic information following the approach described in Section 4.4.1 if these functions are invoked.

There exist six PMU counters for each processor on Juno board, and we randomly select the last one to be used for the Android API trace and the DS. Firstly, the `E` bit of `PMCR_EL0` register is set to enable the PMU. Then, both `PMCNTENSET_EL0` and `PMINTENSET_EL1` registers are set to `0x20` to enable the counter 6 and the overflow interrupt of the counter 6. Next, we set `PMEVTYPER5_EL0` register to `0x80000021` to make the counter 6 count the `BR_RETIRED` event in non-secure EL0. Finally, the counter `PMEVCNTR5_EL0` is set to its maximum value `0xFFFFFFFF`. With this configuration, a secure PMI is routed to EL3 after the execution of the next branch instruction. In the interrupt handler, the `ELR_EL3` register, which is identical to the `PC` of the normal domain, is examined to identify whether the execution of normal domain encounters `ExecuteGotoImpl` or `ExecuteSwitchImpl` function. If true, we fill the semantic gap and fetch the information about the current executing Java method. By the declaring class of the method, we differentiate the Android APIs from the developer defined methods. Before returning to the normal domain, we reset the performance counter to its maximum value to make sure the next execution of a branch instruction leads to an overflow.

4.4.4 The Debug Subsystem

Debugging is another essential approach to learn the behavior of an application. Ninja leverages a secure serial port to connect the board to an external debugging client. There exists two serial port (i.e., `UART0` and `UART1`) in Juno board, and the ATF uses `UART0` as

the debugging input/output of both normal domain and secure domain. To build a secure debugging bridge, Ninja uses `UART1` as the debugging channel and marks it as a secure device by configuring NIC-400 [21]. Alternatively, we can use a USB cable for this purpose. In the DS, an analyst pauses the execution of the target application by the secure NMI or predefined breakpoints and send debugging commands to the board via the secure serial port. Ninja processes the commands and outputs the response to the serial port with a user-friendly format. The information about symbols in both bytecode and machine code are not supported at this moment, and we consider it as our future work.

Single-instruction Stepping

The ARMv8 architecture provides instruction stepping support for the debuggers by the `SS` bit of `MDSCR_EL1` register. Once this bit is set, the CPU generates a software step exception after each instruction is executed, and the highest EL that this exception can be routed is EL2. However, this approach has two fundamental drawbacks: 1) the EL2 is normally prepared for the hardware virtualization systems, which does not satisfy our transparency requirements. 2) The instruction stepping changes the value of `PSTATE`, which is accessible from EL1. Thus, we cannot use the software step exception for the instruction stepping. Another approach is to modify the target application's code to generate a SMC exception after each instruction. Nonetheless, the modification brings the side effect that the self-checking malware may be aware of it.

The PMU event `INST_RETIRED` is fired after the execution of each instruction, and we use this event to implement instruction stepping by using similar approach mentioned in Android API tracing. With the configuration, Ninja pauses the execution of the target after

the execution of each instruction and waits for the debugging commands.

Moreover, Ninja is capable of stepping Java bytecode. Recall that the functions `ExecuteGotoImpl` and `ExecuteSwitchImpl` interpret the bytecode in Java methods. In both functions, a branch instruction is used to switch to the interpretation code of each Java bytecode. Thus, we use `BR_RETIRED` event to trace the branch instructions and firstly ensure the `pc` of normal domain is inside the two interpreter functions. Next, we fill the semantic gap and monitor the value of `dex_pc`. As the change of `dex_pc` value indicates the change of current interpreting bytecode, we pause the system once the `dex_pc` is changed to achieve Java bytecode stepping.

Breakpoints

In ARMv8 architecture, a breakpoint exception is generated by either a software breakpoint or a hardware breakpoint. The execution of `brk` instruction is considered as a software breakpoint while the breakpoint control registers `DBGBCR_EL1` and breakpoint value registers `DBGBVR_EL1` provide support for at most 16 hardware breakpoints. However, similar to the software step exception, the breakpoint exception generated in the normal domain could not be routed to EL3, which breaks the transparency requirement of Ninja. MalT [285] discusses another breakpoint implementation that modifies the target's code to trigger an interrupt. Due to the transparency requirement, we avoid this approach to keep our system transparent against the self-checking malware. Thus, we implement the breakpoint based on the instruction stepping technique discussed above. Once the analyst adds a breakpoint, Ninja stores its address and enable PMU to trace the execution of instructions. If the address of an executing instruction matches the breakpoint, Ninja pauses

the execution and waits for debugging commands. Otherwise, we return to the normal domain and do not interrupt the execution of the target.

Memory Read/Write

Ninja supports memory access with both physical and virtual addresses. The TrustZone technology ensures that EL3 code can access the physical memory of the normal domain, so it is straight forward for Ninja to access memory via physical addresses. Regarding to memory accesses via virtual addresses, we have to find the corresponding physical addresses for the virtual addresses in the normal domain. Instead of manually walk through the page tables, a series of Address Translation (AT) instructions help to translate a 64-bit virtual address to a 48-bit physical address[3] considering the translation stages, ELs and memory attributes. As an example, the `at s12e0r` *addr* instruction performs stage 1 and 2 (if available) translations as defined for EL0 to the 64-bit address *addr*, with permissions as if reading from *addr*. The [47:12] bits of the corresponding physical address are storing in the `PA` bits of the `PAR_EL1` register, and the [11:0] bits of the physical address are identical to the [11:0] bits of the virtual address *addr*. After the translation, Ninja directly manipulates the memory in normal domain according to the debugging commands.

4.4.5 Interrupt Instruction Skid

In ARMv8 manual, the interrupts are referred as asynchronous exceptions. Once an interrupt source is triggered, the CPU continues executing the instructions instead of waiting for the interrupt. Figure 10 shows the interrupt process in Juno board. Assume that an interrupt source is triggered before the `MOV` instruction is executed. The processor then sends the interrupt request to the GIC and continues executing the `MOV` instruction. The

[3]The ARMv8 architecture does not support more bits in the physical address at this moment

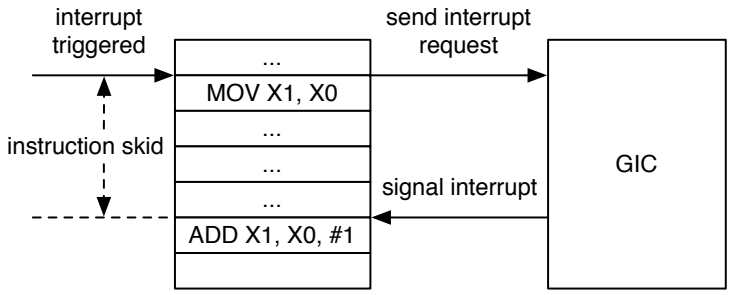


Figure 10: Interrupt Instruction Skid.

GIC processes the requested interrupt according to the configuration, and signals the interrupt back to the processor. Note that it takes GIC some time to finish the process, so some instructions following the `MOV` instruction have been executed when the interrupt arrives the processor. As shown in Figure 10, the current executing instruction is the `ADD` instruction instead of the `MOV` instruction when the interrupt arrives, and the instruction shadow region between the `MOV` and `ADD` instructions is considered as interrupt instruction skid.

The skid problem is a well-known problem [237, 260] and affects Ninja since the current executing instruction is not the one that triggers the PMI when the PMI arrives the processor. Thus, the DS may not exactly step the execution of the processor. Although the skid problem cannot be completely eliminated, the side-effect of the skid does not affect our system significantly, and we provide a detailed analysis and evaluation in Section 4.6.6.

4.5 Transparency

As Ninja is not based on the emulator or other sandboxes, the anti-analysis techniques mentioned in [131, 193, 258] cannot detect the existence of Ninja. Moreover, other anti-debugging techniques like anti-ptrace [281] do not work for Ninja since our analysis does not use ptrace. Nonetheless, Ninja leaves artifacts such as changes of the registers and

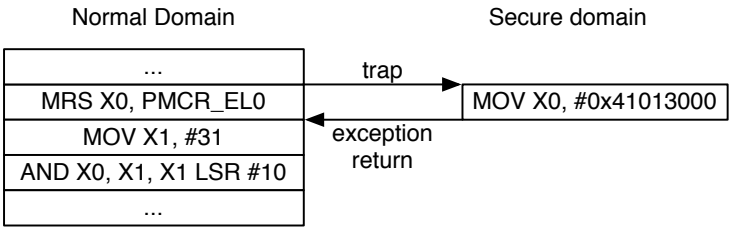


Figure 11: Protect the PMCR_EL0 Register via Traps.

the slow down of the system, which may be detected by the target application. Next, we discuss the mitigation of these artifacts.

4.5.1 Footprints Elimination

Since Ninja works in the secure domain, the hardware prevents the target application from detecting the code or memory usage of Ninja. Moreover, as the ATF restores all the general purpose registers while entering the secure domain and resumes them back while returning to the normal domain, Ninja does not affect the registers used by the target application as well. However, as we use ETM and PMU to achieve the debugging and tracing functions, the modification to the PMU registers and the ETM registers leaves a detectable footprint. In ARMv8, the PMU and ETM registers are accessible via both system-instruction and memory-mapped interfaces.

System-Instruction Interface

The system-instruction interface makes the system registers readable via MRS instruction and writable via MSR instruction. In Ninja, we ensure that the access to the target system registers via these instructions to be trapped to EL3. The TPM bit of the MDCR_EL3 register and the TTA bit of the CPTR_EL3 register help to trap the access to PMU and ETM registers to EL3, respectively; then we achieve the transparency by providing artificial val-

ues to the normal domain. Figure 11 is an example of manipulating the reading to the PMCR_EL0 register and returning the default value of the register. Before the MRS instruction is executed, a trap is triggered to switch to the secure domain. Ninja then analyzes the instruction that triggers the trap and learns that the return value of PMCR_EL0 is stored to the general-purpose register X0. Thus, we put the default value 0x41013000 to the general-purpose register X0 and resume to the normal domain. Note that the PC register of the normal domain should also be modified to skip the MRS instruction. We protect both the registers that we modified (e.g., PMCR_EL0, PMCNTENSET_EL0) and the registers modified by the hardware as a result of our usage (e.g., PMINTENCLR_EL1, PMOVSCLR_EL0).

Memory Mapped Interface

Each of the PMU or ETM related components occupies a distinct physical memory region, and the registers of the component can be accessed via offsets in the region. Since these memory regions do not locate in the DRAM (i.e., main memory), the TrustZone Address Space Controller (TZASC) [31], which partitions the DRAM into secure regions and non-secure regions, cannot protect them directly. Note that this hardware memory region is not initialized by the system firmware by default and the system software such as applications and OSes cannot access it because the memory region is not mapped into the virtual memory. However, advanced malware might remap this physical memory region via functions like mmap and ioremap. Thus, to further defend against these attacks, we intercept the suspicious calls to these functions and redirect the call to return an artificial memory region.

The memory size for both the PMU and ETM memory regions is 64k, and we reserve a

128k memory region on the DRAM to be the artificial PMU and ETM memory. The ATF for Juno board uses the DRAM region 0x880000000 to 0x9fffffff as the memory of the rich OS and the region 0xa00000000 to 0x1000000000 of the DRAM is not actually initialized. Thus, we randomly choose the memory region 0xa00040000 to 0xa00060000 to be the region for artificial memory mapped registers. While the system is booting, we firstly duplicate the values in the PMU and ETM memory regions into the artificial regions. As the function calls are achieved by `bl` instruction, we intercept the call to the interested functions by using PMU to trigger a PMI on the execution of branch instructions and compare the `pc` of the normal domain with the address of these functions. Next, we manipulate the call to these functions by modification to the parameters. Take `ioremap` function as an example. The first parameter of the function, which is stored in the register `X0`, indicates the target physical address, and we modify the value stored at the register to the corresponding address in the artificial memory region. With this approach, the application never reads the real value of PMU and ETM registers, and cannot be aware of Ninja.

4.5.2 Defending Against Timing Attacks

The target application may use the SoC or external timers to detect the time elapsed in the secure domain since the DS affects the performance of the processor and communicates with a human analyst. Note that the TS using ETM does not affect the performance of the processor and thus is immune to the timing attack.

The ARMv8 architecture defines two types of timer components, i.e., the memory-mapped timers and the generic timer registers [20]. Other than these timers, the Juno board is equipped with an additional Real Time Clock (RTC) component PL031 [29] and two dual-timer modules SP804 [25] to measure the time. For each one of these compo-

nents, we manipulate its value to make the time elapsed of Ninja invisible.

Each of the memory-mapped timer components is mapped to a pre-defined memory region, and all these memory regions are writable in EL3. Thus, we record the value of the timer or counter while entering Ninja and restore it before existing Ninja. The RTC and dual-timer modules are also mapped to a writable memory region, so we use a similar method to handle them.

The generic timer registers consist of a series of timer and counter registers, and all of these registers are writable in EL3 except the physical counter register `CNTPCT_EL0` and the virtual counter register `CNTVCT_EL0`. For the writable registers, we use the same approach as handling memory-mapped timers to manipulate them. Although `CNTPCT_EL0` is not directly writable, the ARM architecture requires a memory-mapped counter component to control the generation of the counter value [20]. In the Juno board, the generic counter is mapped to a controlling memory frame 0x2a430000-0x2a43ffff, and writing to the memory address 0x2a430008 updates the value of `CNTPCT_EL0`. The `CNTVCT_EL0` register always holds a value equal to the value of the physical counter register minus the value of the virtual offset register `CNTVOFF_EL2`. Thus, the update to the `CNTPCT_EL0` register also updates the `CNTVCT_EL0` register.

Note that the above mechanism only considers the time consumption of Ninja, and does not take the time consumption of the ATF into account. Thus, to make it more precise, we measure the average time consumption of the ATF during the secure exception handling and minus it while restoring the timer values. Besides the timers, the malware may also leverage the PMU to count the CPU cycles. Thus, Ninja checks the enabled PMU counters and restores their values in a similar way to the writable timers.

Table 9: Comparing with Other Tools. The source lines of code (SLOC) of the TCB is calculated by `sloccount` [255] based on Android 5.1.1 and Linux kernel 3.18.20.

ATF = ARM Trusted Firmware, AOS = Android OS, LK = Linux Kernel

	Ninja	TaintDroid	TaintART	DroidTrace	CrowDroid	DroidScope	CopperDroid	NDroid
No VM/emulator	✓	✓	✓	✓	✓			
No ptrace/strace	✓	✓	✓			✓	✓	✓
No modification to Android	✓			✓	✓	✓	✓	✓
Analyzing native instruction	✓			✓	✓	✓	✓	✓
Trusted computing base	ATF	AOS + LK	AOS + LK	LK	LK	QEMU	QEMU	QEMU
SLOC of TCB (K)	27	56,355	56,355	12,723	12,723	489	489	489

The external timing attack cannot be defended by modifying the local timer since external timers are involved. As the instruction tracing in Ninja is immune to the timing attack, we can use the TS to trace the execution of the target with DS enabled and disabled. By comparing the trace result using the approaches described in BareCloud [141] and MalGene [140], we may identify the suspicious instructions that launch the attack and defend against the attack by manipulating the control flow in EL3 to bypass these instructions. However, the effectiveness of this approach needs to be further studied. Currently, defending against the external timing attack is an open research problem [77, 285].

4.6 Evaluation

To evaluate Ninja, we fist compare it with existing analysis and debugging tools on ARM. Ninja neither involves any virtual machine or emulator nor uses the detectable Linux tools like `ptrace` or `strace`. Moreover, to further improve the transparency, we do not modify Android system software or the Linux kernel. The detailed comparison is listed in Table 9. Since Ninja only relies on the ATF, the table shows that the Trusted Computing Base (TCB) of Ninja is much smaller than existing systems.

4.6.1 Output of Tracing Subsystem

To learn the details of the tracing output, we write a simple Android application that uses Java Native Interface to read the `/proc/self/status` file line by line (which can be further used to identify whether `ptrace` is enabled) and outputs the content to the console. We use instruction trace of the TS to trace the execution of the application, and also measure the time usage. The status file contains 38 lines in total, and it takes about $0.22\ ms$ to finish executing. After the execution, the ETF contains $9.92\ KB$ encoded trace data, and the datarate is approximately $44.03\ MB/s$. Next, we use `ptm2human` [112] to decode the data, and the decoded trace data contains 1341 signpost instructions (80 in our custom native library and the others in `libc.so`). By manually introspect the signpost instructions in our custom native library, we can rebuild the whole execution control flow. To reduce the storage usage of the ETM, we can use real-time continuous export via either a dedicated trace port capable of sustaining the bandwidth of the trace or an existing interface on the SoC (e.g., a USB or other high-speed port) [26].

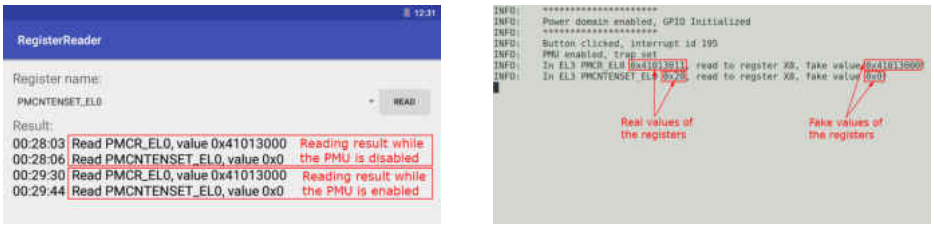
4.6.2 Tracing and Debugging Samples

We randomly pickup two samples `ActivityLifecycle1` and `PrivateDataLeak3` from DroidBench [83] project and use Ninja to analyze them. We choose these two specific samples since they exhibit representative malicious behavior like leaking sensitive information via local file, text message, and network connection.

Analyzing *ActivityLifecycle1*. To get an overview of the sample, we first enable the Android API tracing feature to inspect the APIs that read sensitive information (source) and APIs that leak information (sink), and find a suspicious API call sequence. In the sequence,

the method `TelephonyManager.getDeviceId` and method `HttpURLConnection.connect` are invoked in turn, which indicates a potential flow that sends IMEI to a remote server. As we know the network packets are sent via the system call `sys_sendto`, we attempt to intercept the system call and analyze the parameters of the system call. In Android, the system calls are invoked by corresponding functions in `libc.so`, and we get the address of the function for the system call `sys_sendto` by disassembling `libc.so`. Thus, we use Ninja to set a breakpoint at the address, and the second parameter of the system call, which is stored in register `X1`, shows that the sample sends a 181 bytes buffer to a remote server. Then, we output the memory content of the buffer and find that it is a `HTTP GET` request to host `www.google.de` with path `/search?q=353626078711780`. Note that the digits in the path is exactly the IMEI of the device.

Analyzing *PrivateDataLeak3*. Similar to the previous analysis, the Android API tracing helps us to find a suspicious API call sequence consisting of the methods `TelephonyManager.getDeviceId`, `Context.openFileOutput`, and `SmsManager.sendTextMessage`. As the Android uses the system calls `sys_openat` to open a file and `sys_write` to write a file, we set breakpoints at the address of these calls. Note that the second parameter of `sys_openat` represents the full path of the target file and the second parameter of `sys_write` points to a buffer writing to a file. Thus, after the breakpoints are hit, we see that sample writing IMEI 353626078711780 to the file `/data/data/de.ecspride/files/out.txt`. The API `SmsManager.sendTextMessage` uses `binder` to achieve IPC with the lower-layer `SmsService` in Android system, and the semantics of the IPC is described in CopperDroid [245]. By intercepting the system call `sys_ioctl` and following the semantics, we finally find the target of the text message "+49" and the content of the message 353626078711780.



(a) Reading PMU Register in an Application.

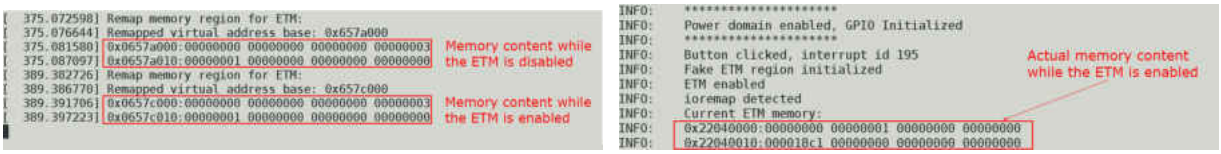
(b) EL3 Output in the Secure Console.

Figure 12: Accessing System Instruction Interface.

4.6.3 Transparency Experiments

Accessing System Instruction Interface

To evaluate the protection mechanism of the system instruction interface, we write an Android application that reads the `PMCR_EL0` and `PMCNTENSET_EL0` registers via `MRS` instruction. The values of these two registers represent whether a performance counter is enabled. We first use the application to read the registers with Ninja disabled, and the result is shown in the upper rectangle of Figure 12a. The last bit of the `PMCR_EL0` register and the value of the `PMCNTENSET_EL0` register are 0, which means that all the performance counters are disabled. Then we press a GPIO button to enable the Android API tracing feature of Ninja and read the registers again. From the console output shown in Figure 12b, we see that the access to the registers is successfully trapped into EL3. And the output shows that the real values of the `PMCR_EL0` and `PMCNTENSET_EL0` registers are 0x41013011 and 0x20, respectively, which indicates that the counter `PMEVCNTR5_EL0` is enabled. However, the lower rectangle in Figure 12a shows that the value of the registers fetched by the application keep unchanged. This experiment shows that Ninja effectively eliminates the footprint on the system instruction interface.



(a) Reading ETM Memory Region. (b) EL3 Output in the Secure Console.

Figure 13: Memory Mapped Interface.

4.6.4 Accessing Memory Mapped Interface

In this section, we take `ioremap` function as an example to evaluate whether the interception to the memory-mapping functions works. As the `ioremap` function can be called only in the kernel space, we write a kernel module that remaps the memory region of the ETM by the `ioremap` function, and print the content of the first 32 bytes in the region. Similar to the approach discussed above, we first load the kernel module with Ninja disabled, and the output is shown in the upper rectangle in Figure 13a. Note that the 5th to the 8th bytes are mapped as the `TRCPRGCTLR` register and the `EN` bit, which indicates the status of the ETM, is the last bit of the register. In the upper rectangle, the `EN` bit 0 shows that the ETM is disabled. Next, we enable the instruction tracing feature of Ninja and reload the kernel module. The lower rectangle in Figure 13a shows that the content of the memory fetched by the module remains the same. However, in the Figure 13b, the output from EL3 shows that the memory of the ETM has changed. This experiment shows that we successfully hide the ETM status change to the normal domain, and Ninja remains transparent.

Adjusting the Timers

To evaluate whether our mechanism that modifies the local timers works, we write a simple application that launches a dummy loop for 1 billion times, and calculate the exe-

Table 10: The TS Performance Evaluation Calculating 1 Million Digits of π.

	Mean	STD	# Slowdown
Base: Tracing disabled	2.133 s	0.69 ms	
Instruction tracing	2.135 s	2.79 ms	$\sim$ 1x
System call tracing	2.134 s	5.13 ms	$\sim$ 1x
Android API tracing	149.372 s	1287.88 ms	$\sim$70x

cution time of the loop by the return values of the API call `System.currentTimeMillis()`. In the first experiment, we record the execution time with Ninja disabled, and the average time for 30 runs is 53.16s with a standard deviation 2.97s. In the second experiment, we enable the debugging mode of Ninja and pause the execution during the loop by pressing the GPIO button. To simulate the manual analysis, we send a command `rr` to output all the general purpose registers and then read them for 60s. Finally, a command `c` is sent to resume the execution of the target. We repeat the second experiment with the timer adjusting feature of Ninja enabled and disabled for 30 times each, and record the execution time of the loop. The result shows that the average execution time with timer adjusting feature disabled is 116.33s with a standard deviation 2.24s, and that with timer adjusting feature enabled is 54.33s with a standard deviation 3.77s. As the latter result exhibits similar execution time with the original system, the malware cannot use the local timer to detect the presence of the debugging system.

4.6.5 Performance Evaluation

In this section, we evaluate the performance overhead of the trace subsystem due to its automation characteristic. Performance overhead of the debugging subsystem is not noticed by an analyst in front of the command console, and the debugging system is designed with human interaction.

Table 11: The TS Performance Evaluation with CF-Bench [61].

	Native Scores			Java Scores			Overall Scores		
	Mean	STD	Slowdown	Mean	STD	Slowdown	Mean	STD	Slowdown
Base: Tracing disabled	25380	1023		18758	1142		21407	1092	
Instruction tracing	25364	908	$\sim$ 1x	18673	1095	$\sim$ 1x	21349	1011	$\sim$ 1x
System call tracing	25360	774	$\sim$ 1x	18664	1164	$\sim$ 1x	21342	911	$\sim$ 1x
Android API tracing	6452	24	$\sim$ 4x	122	4	$\sim$ 154x	2654	11	$\sim$ 8x

To learn the performance overhead on the Linux binaries, we build an executable that using an open source π calculation algorithm provided by the GNU Multiple Precision Arithmetic Library [251] to calculate 1 million digits of the π for 30 times with the tracing functions disabled and enabled, and the time consumption is shown in Table 10. Since we leverage ETM to achieve the instruction tracing and system call tracing, the experiment result shows that the ETM-based solution has negligible overhead — less than 0.1%. In the Android API tracing, the overhead is about 70x. This overhead is mainly due to the frequent domain switch during the execution and bridging the semantic gap. To reduce the overhead, we can combine ETM instruction trace with data trace, and leverage the trace result to rebuild the semantic information and API usage offline.

To measure the performance overhead on the Android applications, we use CF-Bench [61] downloaded from Google Play Store. The CF-Bench focuses on measuring both the Java performance and native performance in Android system, and we use it to evaluate the overhead for 30 times. The result in Table 11 shows that the overheads of instruction tracing and system call tracing are sufficiently small to ignore. The Android API tracing brings 4x slowdown on the native score and 154x slowdown on the Java score, and the overall slowdown is 8x. Note that we make these benchmarks to be executed only on Cortex-A57 core 0 by setting their CPU affinity mask to 0x1 since Ninja only stays in that core.

Table 12: Instructions in the Skid Shadow with Representative PMU Events.

Event Number	Event Description	# of Instructions	
		Mean	STD
0x81-0x8F	Exception related events that firing after taking exceptions	0	0
0x11	CPU cycle event that firing after each CPU cycle	2.73	2.30
0x08	Instruction retired event that firing after executing each instruction	6.03	4.99

4.6.6 Skid Evaluation

In this subsection, we evaluate the influence of the skid problem to Ninja. Since the instruction tracing, system call tracing, and memory read/write do not involve PMI, these functionalities are not affected by the skid problem. In ART, each bytecode is interpreted as an array of machine code. Our bytecode stepping mechanism recognizes the corresponding bytecode once it is executing any machine code in the array, i.e., the skid problem affects the bytecode stepping if and only if the instruction shadow covers all the machine code for a bytecode. We evaluate the listed 218 bytecode opcode [95] on the Android official website, and it shows that the shadow region cannot cover the machine code for any of them. Thus, the bytecode stepping does not suffer from the skid problem. For a similar reason, the skid problem has no influence on the Android API tracing.

However, the native code stepping and the breakpoint are still affected, and both of them use instruction retired event to overflow the counter. Since the skid problem is due to the delay between the interrupt request and the interrupt arrival, we first use PMU counter to measure this delay by CPU cycles. Similar with the instruction stepping, we make the PMU counter to count `CPU_CYCLES` event and initialize the value of the counter

to its maximum value. Then, the counter value after switching into EL3 is the time delay of the skid in CPU cycles. The results of 30 experiments show that the delay is about 106.3 CPU cycles with a standard deviation 2.26. As the frequency of our CPU is 1.15GHz, the delay is about 0.09μs. We also evaluate the number of instructions in the skid shadow with some representative PMU events. For each event, we trigger the PMI for 30 times and calculate the mean and standard deviation of the number of instructions in the shadow. Table 12 shows the result with different PMU events. Unlike the work described in [237], the exception related events exhibits no instruction shadow in our platform, and we consider it is caused by different ARM architectures. It is worth noting that the number of instructions in the skid shadow of the CPU cycle event is less than the instruction retired event. However, using the CPU cycle event may lead to multiple PMIs for a single instruction since the execution of a single instruction may need multiple CPU cycles, which introduces more performance overhead but with more fine-grained instruction-stepping. In practice, it is a trade off between the performance overhead and the debugging accuracy, and we can use either one based on the requirement.

CHAPTER 5 UNDERSTANDING THE SECURITY OF ARM DEBUGGING FEATURES

5.1 Introduction

Although the debugging architecture and authentication signals have been presented for years, the security of them is under-examined by the community since it normally requires physical access to use these features in the traditional debugging model. However, ARM introduces a new debugging model that requires no physical access since ARMv7 [18]. As shown in the left side of Figure 14, in the traditional debugging model, an off-chip debugger connects to an on-chip Debug Access Port (DAP) via the JTAG interface, and the DAP further helps the debugger to debug the on-chip processors. In this model, the off-chip debugger is the debug host, and the on-chip processors are the debug target. The right side of Figure 14 presents the new debugging model introduced since ARMv7. In this model, a memory-mapped interface is used to map the debug registers into the memory so that the on-chip processor can also access the DAP. Consequently, an on-chip processor can act as a debug host and debug another processor (the debug target) on the same chip; we refer to this debugging model as the inter-processor debugging model. Nevertheless, ARM does not provide an upgrade on the privilege management mechanism for the new debugging model, and still uses the legacy debug authentication signals in the inter-processor debugging model, which exacerbates our concern on the security of the debugging features.

We dig into the ARM debugging architecture to acquire a comprehensive understanding of the debugging features, and summarize the security implications. We note that the debug authentication signals only take the privilege mode of the debug target into account

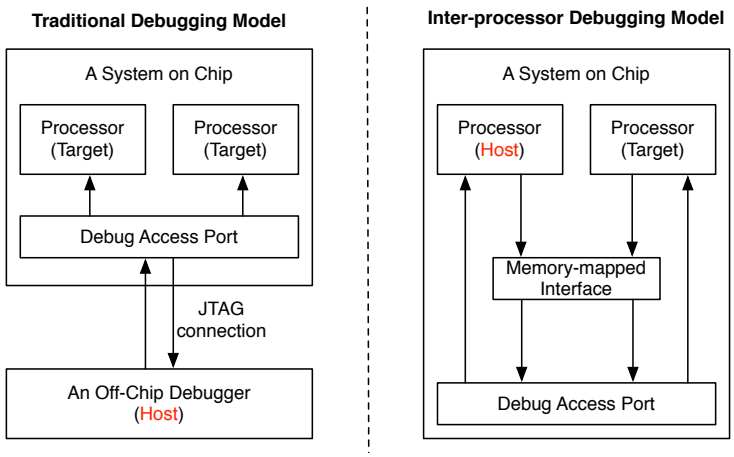


Figure 14: Debug Models in ARM Architecture.

and ignore the privilege mode of the debug host. It works well in the traditional debugging model since the debug host is an off-chip debugger in this model, and the privilege mode of the debug host is not relevant to the debug target. However, in the inter-processor debugging model, the debug host and debug target locate at the same chip and share the same resource (e.g., memory and registers), and reusing the same debug authentication mechanism leads to the privilege escalation via misusing the debugging features. With help of another processor, a low-privilege processor can obtain arbitrary access to the high-privilege resource such as code, memory, and registers. Note that the low-privilege in this paper mainly refers to the kernel-level privilege, while the high-privilege refers to the secure privilege levels provided by TrustZone [31] and the hypervisor-level privilege.

This privilege escalation depends on the debug authentication signals. However, ARM does not provide a standard mechanism to control these authentication signals, and the management of these signals highly depends on the System-on-Chip (SoC) manufacturers. Thus, we further conduct an extensive survey on the debug authentication signals in different ARM-based platforms. Specifically, we investigate the default status and the

management mechanism of these signals on the devices powered by various SoC manufacturers, and the target devices cover four product domains including development boards, Internet of Things (IoT) devices, commercial cloud platforms, and mobile devices.

In our investigation, we find that the debug authentication signals are fully or partially enabled on the investigated platforms. Meantime, the management mechanism of these signals is either undocumented or not fully functional. Based on this result, we craft a novel attack scenario, which we call Nailgun[4]. Nailgun works on a processor running in a low-privilege mode and accesses the high-privilege content of the system without restriction via the aforementioned new debugging model. Specifically, with Nailgun, the low-privilege processor can trace the high-privilege execution and even execute arbitrary payload at a high-privilege mode. To demonstrate our attack, we implement Nailgun on commercial devices with different SoCs and architectures, and the experiment results show that Nailgun is able to break the privilege isolation enforced by the ARM architecture. Our experiment on Huawei Mate 7 also shows that Nailgun can leak the fingerprint image stored in TrustZone from the commercial mobile phones. In addition, we present potential countermeasures to our attack in different perspectives of the ARM ecosystem. Note that the debug authentication signals cannot be simply disabled to avoid the attack, and we will discuss this in Section 5.5.

The hardware debugging features have been deployed to the modern processors for years, and not enough attention is paid to the security of these features since they require physical access in most cases. However, it turns out to be vulnerable in our analysis when the multiple-processor systems and inter-processor debugging model are involved. We

[4]Nailgun is a tool that drives nails through the wall—breaking the isolation

consider this as a typical example in which the deployment of new and advanced systems impacts the security of a legacy mechanism. The intention of this paper is to rethink the security design of the debugging features and motivate the researchers/developers to draw more attention to the "known-safe" or "assumed-safe" components in the existing systems.

5.2 Security Implications of the Debugging Architecture

As mentioned in Section 2.3, non-invasive debugging and invasive debugging are available in ARM architecture. In this section, we carefully investigate the non-invasive and invasive debugging mechanisms documented in the Technique Reference Manuals (TRM) [18, 20], and reveal the vulnerability and security implications indicated by the manual. Note that we assume the required debug authentication signals are enabled in this section, and this assumption is proved to be reasonable and practical in Section 5.3.

5.2.1 Non-invasive Debugging

The non-invasive debugging does not allow to halt a processor and introspect the state of the processor. Instead, non-invasive features such as the Performance Monitor Unit (PMU) and Embedded Trace Macrocell (ETM) are used to count the processor events and trace the execution, respectively.

In the ARMv8 architecture, the PMU is controlled by a group of registers that are accessible in non-secure EL1. However, we find that ARM allows the PMU to monitor the events fired in EL2 even when the `NIDEN` signal is disabled [5]. Furthermore, the PMU can monitor the events fired in the secure state including EL3 with the `SPNIDEN` signal enabled. In other words, an application with non-secure EL1 privilege is able to monitor the events

[5] In ARMv7, `NIDEN` is required to make PMU monitor the events in non-secure state.

fired in EL2 and the secure state with help of the debug authentication signals. The TPM bit of the MDCR register is introduced in ARMv8 to restrict the access to the PMU registers in low ELs. However, this restriction is only applied to the system register interface but not the memory-mapped interface [20].

The ETM traces the instructions and data streams of a target processor with a group of configuration registers. Similar to the PMU, the ETM is able to trace the execution of the non-secure state (including EL2) and the secure state with the NIDEN and SPNIDEN signals, respectively. However, it only requires non-secure EL1 to access the configuration registers of the ETM. Similar to the aforementioned restriction on the access to the PMU registers, the hardware-based protection enforced by the TTA bit of the CPTR register is also applied to only the system register interface [20].

In conclusion, the non-invasive debugging feature allows the application with a low privilege to learn information about the high-privilege execution.

Implication 1: An application in the low-privilege mode is able to learn information about the high-privilege execution via PMU and ETM.

5.2.2 Invasive Debugging

The invasive debugging allows an external debugger to halt the target processor and access the resources on the processor via the debugging architecture. Figure 15 shows a typical invasive debugging model. In the scenario of invasive debugging, we have an external debugger (HOST) and the debug target processor (TARGET). To start the debugging, the HOST sends a debug request to the TARGET via the ECT. Once the request is handled, the communication between the HOST and TARGET is achieved via the instruction transfer-

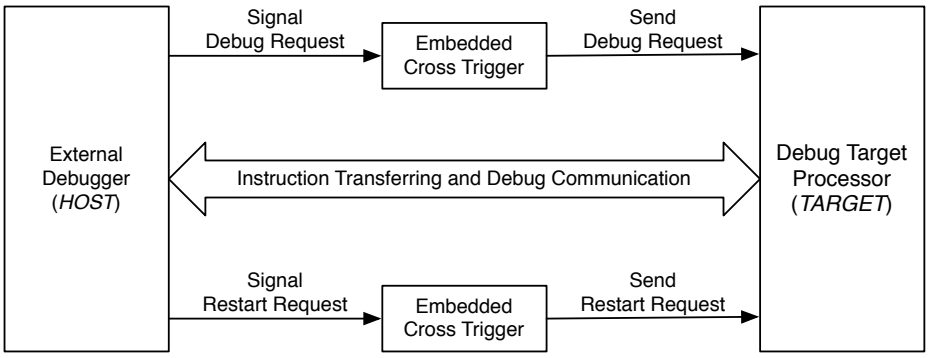


Figure 15: Invasive Debugging Model.

ring and data communication channel provided by the debugging architecture. Finally, the restart request is used to end the debugging session. In this model, since the HOST is always considered as an external debugging device or a tool connected via the JTAG port, we normally consider it requires physical access to debug the TARGET. However, ARM introduces an inter-processor debugging model that allows an on-chip processor to debug another processor on the same chip without any physical access or JTAG connection since ARMv7. Furthermore, the legacy debug authentication signals, which only consider the privilege mode of the TARGET but ignore the privilege mode of the HOST, are used to conduct the privilege control of the inter-processor debugging model. In this section, we discuss the security implications of the inter-processor debugging under the legacy debug authentication mechanism.

Entering and Existing Debug State

To achieve the invasive debugging in the TARGET, we need to make the TARGET run in the debug state. The processor running in the debug state is controlled via the external debug interface, and it stops executing instructions from the location indicated by the program counter. There are two typical approaches to make a processor enter the debug

state: executing an HLT instruction on the processor or sending an external debug request via the ECT.

The HLT instruction is widely used as a software breakpoint, and executing an HLT instruction directly causes the processor to halt and enter the debug state. A more general approach to enter the debug state is to send an external debug request via the ECT. Each processor in a multiple-processor system is embedded with a separated CTI (i.e., interface to ECT), and the memory-mapped interface makes the CTI on a processor available to other processors. Thus, the HOST can leverage the CTI of the TARGET to send the external debug request and make the TARGET enter the debug state. Similarly, a restart request can be used to exit the debug state.

However, the external debug request does not take the privilege of the HOST into consideration; this design allows a low-privilege processor to make a high-privilege processor enter the debug state. For example, a HOST running in non-secure state can make a TARGET running in secure state enter the debug state with the SPIDEN enabled. Similarly, a HOST in non-secure EL1 can halt a TARGET in EL2 with the DBGEN enabled.

Implication 2: A low-privilege processor can make an arbitrary processor (even a high-privilege processor) enter the debug state via ECT..

Debug Instruction Transfer/Communication

Although the normal execution of a TARGET is suspended after entering the debug state, the External Debug Instruction Transfer Register (EDITR) enables the TARGET to execute instructions in the debug state. Each processor owns a separated EDITR register, and writing an instruction (except for special instructions like branch instructions) to this register

when the processor is in the debug state makes the processor execute it.

Meantime, the Debug Communication Channel (DCC) enables data transferring between a HOST in the normal state and a TARGET in the debug state. In ARMv8 architecture, three registers exist in the DCC. The 32-bit DBGDTRTX register is used to transfer data from the TARGET to the HOST, while the 32-bit DBGDTRRX register is used to receive data from the HOST. Moreover, the 64-bit DBGDTR register is available to transfer data in both directions with a single register.

We note that the execution of the instruction in the EDITR register only depends on the privilege of the TARGET and ignores the privilege of the HOST, which actually allows a low-privilege processor to access the high-privilege resource via the inter-processor debugging. Assume that the TARGET is running in the secure state and the HOST is running in the non-secure state, the HOST is able to ask the TARGET to read the secure memory via the EDITR register and further acquire the result via the DBGDTRTX register.

Implication 3: In the inter-processor debugging, the instruction execution and resource access in the TARGET does not take the privilege of the HOST into account..

Privilege Escalation

The *Implication 2* and *Implication 3* indicate that a low-privilege HOST can access the high-privilege resource via a high-privilege TARGET. However, if the TARGET remains in a low-privilege mode, the access to the high-privilege resource is still restricted. ARM offers an easy way to escalate privilege in the debug state. The dcps1, dcps2, and dcps3 instructions, which are only available in debug state, can directly promote the exception level of a processor to EL1, EL2, and EL3, respectively.

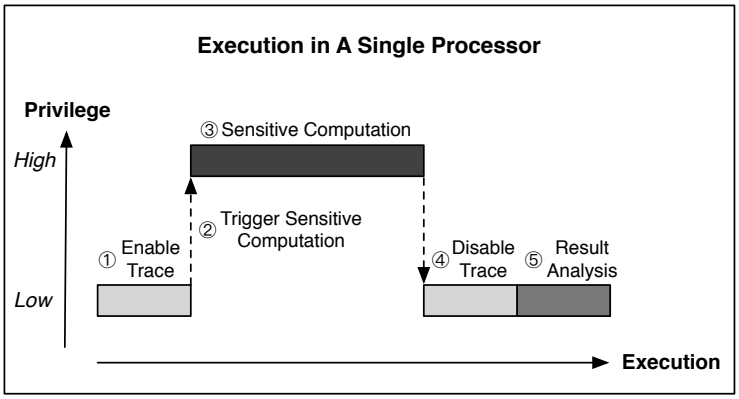


Figure 16: Violating the Isolation via Non-Invasive Debugging.

The execution of the `dcps` instructions has no privilege restriction, i.e., they can be executed at any exception level regardless of the secure or non-secure state. This design enables a processor running in the debug state to achieve an arbitrary privilege without any restriction.

Implication 4: The privilege escalation instructions enable a processor running in the debug state to gain a high privilege without any restriction..

5.2.3 Summary

Both the non-invasive and invasive debug involve the design that allows an external debugger to access the high-privilege resource while certain debug authentication signals are enabled, and the privilege mode of the debugger is ignored. In the traditional debugging model that the `HOST` is off-chip, this is reasonable since the privilege mode of the off-chip platform is not relevant to that of the on-chip platform where the `TARGET` locates. However, since ARM allows an on-chip processor to act as an external debugger, simply reusing the rules of the debug authentication signals in the traditional debugging model makes the on-chip platform vulnerable.

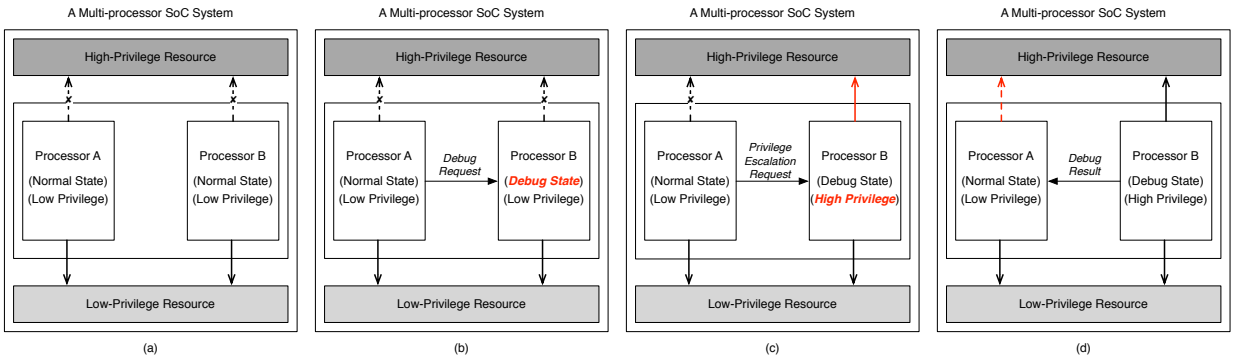


Figure 17: Privilege Escalation in A Multi-processor SoC System via Invasive Debugging.

Non-invasive Debugging: Figure 16 shows an idea of violating the privilege isolation via the non-invasive debugging. The execution of a single processor is divided into different privilege modes, and isolations are enforced to protect the sensitive computation in the high-privilege modes from the low-privilege applications. However, a low-privilege application is able to violate this isolation with some simple steps according to *Implication 1*. Step ① in Figure 16 enables the ETM trace from the low-privilege application to prepare for the violation. Next, we trigger the sensitive computation to switch the processor to a high-privilege mode in step ②. Since the ETM is enabled in step ①, the information about the sensitive computation in step ③ is recorded. Once the computation is finished, the processor returns to a low-privilege mode and the low-privilege application disables the trace in step ④. Finally, the information about the sensitive computation is revealed via analyzing the trace output in step ⑤.

Invasive Debugging: In regard to the invasive debugging, the *Implications 2-4* are unneglectable in the inter-processor debugging model since the `HOST` and `TARGET` work in the same platform and share the same resource (e.g., memory, disk, peripheral, and etc.). As described in Figure 17(a), the system consists of the high-privilege resource, the low-

privilege resource, and a dual-core cluster. By default, the two processors in the cluster can only access the low-privilege resource. To achieve the access to the high-privilege resource, the processor A acts as an external debugger and sends a debug request to the processor B. In Figure 17(b), the processor B enters the debug state due to the request as described in *Implication 2*. However, neither of the processors is able to access the high-privilege resource since both of them are still running in the low-privilege mode. Next, as shown in Figure 17(c), the processor A makes the processor B execute a privilege escalation instruction. The processor B then enters the high-privilege mode and gains access to the high-privilege resource according to *Implication 4*. At this moment, accessing the high-privilege resource from the processor A is still forbidden. Finally, since the processor A is capable of acquiring data from the processor B and the processor B can directly access the high-privilege resource, as indicated by *Implication 3*, the low-privilege processor A actually gains an indirect access to the high-privilege resource as shown in Figure 17(d).

Unlike the traditional debugging model, the non-invasive debugging in Figure 16 and invasive debugging in Figure 17 require no physical access or JTAG connection.

5.3 Debug Authentication Signals in Real-World Devices

The aforementioned isolation violation and privilege escalation occur only when certain debug authentication signals are enabled. Thus, the status of these signals is critical to the security of the real-world devices, which leads us to perform an investigation on the default status of the debug authentication signals in real-world devices. Moreover, we are also interested in the management mechanism of the debug authentication signals deployed on the real-world devices since the mechanism may be used to change the status

of the signals at runtime. Furthermore, as this status and management mechanism highly depend on the SoC manufacturers and the OEMs, we select various devices powered by different SoCs and OEMs as the investigation target. To be comprehensive, we also survey the devices applied in different product domains including development boards, Internet of Things (IoT) devices, commercial cloud platforms, and mobile devices. We discuss our choices on the target devices in Section 5.3.1, and present the results of the investigation in Section 5.3.2 and Section 5.3.3.

5.3.1 Target Devices

Development Boards

The ARM-based development boards are broadly used to build security-related analysis systems [52, 103, 108, 241, 290]. However, the security of the development board itself is not well-studied. Therefore, we select the widely used development board [52, 241, 290], i.MX53 Quick Start Board (QSB) [182], as our analysis object. As a comparison, the official Juno Board [28] released by ARM is also studied in this paper.

IoT Devices

The low power consumption makes the ARM architecture to be a natural choice for the Internet of Things (IoT) devices. Many traditional hardware vendors start to provide the ARM-based smart home solutions [16, 164, 212], and experienced developers even build their own low-cost solutions based on cheap SoCs [104]. As a typical example, the Raspberry PI 3 [203], over $9,000,000$ units of which have been sold till March 2018 [202], is selected as our target.

Commercial Cloud Platforms:

Table 13: Debug Authentication Signals on Real Devices.

Category	Company	Platform / Device	SoC		Debug Authentication Signals			
			Company	Name	DBGEN	NIDEN	SPIDEN	SPNIDEN
Development Boards	ARM	Juno r1 Board	ARM	Juno	✔	✔	✔	✔
	NXP	i.MX53 QSB	NXP	i.MX53	✖	✔	✖	✖
IoT Devices	Raspberry PI	Raspberry PI 3 B+	Broadcom	BCM2837	✔	✔	✔	✔
Commercial Cloud Platforms	miniNodes	64-bit ARM miniNode	Huawei	Kirin 620	✔	✔	✔	✔
	Packet	Type 2A Server	Cavium	ThunderX	✔	✔	✔	✔
	Scaleway	ARM C1 Server	Marvell	Armada 370/XP	✔	✔	✔	✔
Mobile Devices	Google	Nexus 6	Qualcomm	Snapdragon 805	✖	✔	✖	✖
	Samsung	Galaxy Note 2	Samsung	Exynos 4412	✔	✔	✖	✖
	Huawei	Mate 7	Huawei	Kirin 925	✔	✔	✔	✔
	Motorola	E4 Plus	MediaTek	MT 6737	✔	✔	✔	✔
	Xiaomi	Redmi 6	MediaTek	MT 6762	✔	✔	✔	✔

The Cloud Computing area is dominated by the x86 architecture, however, the benefit of the high-throughput computing in ARM architecture starts to gain the attention of big cloud providers including Microsoft [267]. Although most of the ARM-based cloud servers are still in test, we use the publicly available ones including miniNodes [165], Packet [188], and Scaleway [216] to conduct our analysis.

Mobile Devices:

Currently, most mobile devices in the market are powered by ARM architecture, and the mobile device vendors build their devices based on the SoCs provided by various SoC manufacturers. For example, Huawei and Samsung design Kirin [106] and Exynos [213] SoCs for their own mobile devices, respectively. Meantime, Qualcomm [200] and MediaTek [163] provide SoCs for various mobile device vendors [167, 168, 274]. Considering both the market share of the mobile vendors [239] and the variety of the SoCs, we select Google Nexus 6, Samsung Galaxy Note 2, Huawei Mate 7, Motorola E4 Plus, and Xiaomi Redmi 6 as our analysis targets.

5.3.2 Status of the Authentication Signals

The Debug Authentication Status Register (DBGAUTHSTATUS) is a read-only register that is accessible in EL1, and the bits[0:7] of this register reflect the status of the authentication signals. For the target devices, we build a Loadable Kernel Module (LKM) to read the status of the debug authentication signals via this register. However, some stock ROMs in the mobile devices forbid the load of LKM. In that case, we obtain the kernel source code of the stock ROM and recompile a kernel image with LKM enabled option. The recompiled image is then flashed back to the device to conduct the investigation. Note that we make no change to other functionalities in the kernel, and the kernel replacement does not affect the status of the authentication signals.

Table 13 summarizes the default status of the debug authentication signals in the tested devices. On the Juno board, which is designed only for development purpose, the debug authentication signals are all enabled by default. However, we are surprised to find that all the debug authentication signals are enabled by default on the commercial devices like Raspberry PI 3 Model B+, Huawei Mate 7, Motorola E4 Plus, and Xiaomi Redmi. Moreover, all the investigated cloud platforms also enable all these signals. The results on other platforms show that the debug authentication signals are partially enabled by default in the tested mobile devices.

For the mobile phones that enable SPNIDEN and SPIDEN, we also investigate the usage of the TrustZone on these devices. According to [11, 102, 215], the Huawei Mate 7, Motorola E4 Plus and Xiaomi Redmi 6 leverage TrustZone to enforce a hardware-level protection on the collected fingerprint image. By manually introspect the binary image

of the TEE in Huawei Mate 7, we also find that there exists an encryption engine inside the TEE. The TEE image of Motorola E4 Plus and Xiaomi Redmi 6 indicate that both of them use ARM Trusted Firmware (ATF) [30] as the TEE OS. The ATF provides support for both trusted boot and trusted apps, and we also find a potential secure patching module in these binaries. In the TEE image of Xiaomi Redmi 6, we identify a large array with pairs of file names and 128-bit checksums, which may be used to verify the integrity of the system files.

5.3.3 Management of the Authentication Signals

To understand the deployed signal management mechanism, we collect information from the publicly available TRMs and the source code released by the hardware vendors. The signal management mechanism on Juno board and i.MX53 QSB is partially documented in the TRMs, and we have also identified some potential-related code in the kernel source code of Motorola Nexus 6 and Huawei Mate 7. In regard to the other platforms, the signal management mechanism cannot be identified from the publicly available TRMs and released source code.

What we learned from the TRMs:

NXP i.MX53 Quick Start Board (QSB). According to the publicly available TRM of i.MX53 SoC [181], the `DBGEN` signal is controlled by the `DBGEN` bit of the `ARM_GPC` register located at memory address `0x63FA0004`, and no privilege requirement is specified for the access to this register. The management of other debug authentication signals is not documented. In the further experiment, we find that the `SPIDEN` and `SPNIDEN` signals can be controlled via the JTAG port. Once we use the JTAG to connect to the board via additional

debugging software (ARM DS-5 [24] or OpenOCD [186]), the SPIDEN and SPNIDEN signals are directly enabled. Note that this mechanism actually breaks ARM's design purpose since it allows a debugger to enable the debug authentication signals which are design to restrict the usage of the debugger.

ARM Juno r1 Board. As an official development platform released by ARM, the management mechanism of the debug authentication signals is well-documented in the TRM of Juno Board [28]. Developers can control the signal via the debug authentication register in the System Configuration Controller (SCC) or the System Security Control (SSC) registers. The SCC is actually managed by a text file in a configuration MircoSD card and the configurations on the card are loaded by the motherboard micro-controller firmware during the early board setup; modification to the text file becomes effective after a reboot. This configuration MircoSD card is not available to the on-chip OS and can be mounted to a remote PC via a dedicated USB cable. In contrast, the SSC registers can be modified at runtime, and they can only be accessed when the processor is running in the secure state. In our experiment, we find that the debug authentication register in the SCC can only be used to manage the SPIDEN and SPNIDEN signals. Clearing the bit 0 of the register, which is documented as "Global External Debug Enable" bit, does not disable any of the debug authentication signals. Similarly, the SSC registers can control the status of the SPIDEN and SPNIDEN signals, but the modification to the DBGEN and NIDEN signals does not work. Unlike the aforementioned i.MX53 QSB, connecting to the external debugging software via JTAG will not enable the SPIDEN and SPNIDEN signals.

What we learned from the source code:

Motorola Nexus 6. We check the kernel source code for Motorola Nexus 6 provided by Android Open Source Project (AOSP) and find that the debug authentication signals are controlled by a CoreSight fuse [218] at address `0xFC4BE024`. Since the fuse is considered as a One-Time Programmable (OTP) device, directly writing to the corresponding memory fails without providing any error messages.

Huawei Mate 7. The kernel source code for Huawei Mate 7 is released at Huawei Open Source Release Center [110]. From the source code, we find that the `DBGEN` signal is controlled by the register located at address `0xFFF0A82C`. However, directly read/write this register leads to a critical fault that makes the phone to reboot. We consider that Huawei has adopted additional protection to prevent the access to this register for security concerns.

5.3.4 Summary

Our investigation shows that the debug authentication signals are fully or partially enabled on all the tested devices by default, which makes them vulnerable to the aforementioned isolation violation and privilege escalation. Moreover, there is no publicly available management mechanism for these signals on all tested devices except for development boards, and the documented management mechanism of development boards is either incomplete (i.MX53 QSB) or not fully functional (Juno Board). On the one hand, the unavailable management mechanism may help to prevent malicious access to the debug authentication signals. On the other hand, it also stops the user to disable the debug authentication signals for defense purpose.

5.4 Nailgun Attack

To verify the security implications concluded in Section 5.2 and the findings about the debug authentication signals described in Section 5.3, we craft an attack named Nailgun and implement it in several different platforms. Nailgun misuses the non-invasive and invasive debugging features in the ARM architecture, and gains the access to the high-privilege resource from a low-privilege mode. To further understand the attack, we design two attacking scenarios for non-invasive and invasive debugging, respectively. With the non-invasive debugging feature, Nailgun is able to infer the AES encryption key, which is isolated in EL3, via executing an application in non-secure EL1. In regard to the invasive debugging feature, Nailgun demonstrates that an application running in non-secure EL1 can execute arbitrarily payloads in EL3. To learn the impact of Nailgun on real-world devices, we show that Nailgun can be used to extract the fingerprint image protected by TEE in Huawei Mate 7. Similar attacks can be launched to attack EL2 from EL1. Since there are three major ARM architectures (i.e., ARMv7, 32-bit ARMv8, and 64-bit ARMv8), we also implement Nailgun on these different architectures and discuss the differences in implementations.

5.4.1 Threat Model and Assumptions

In our attack, we make no assumption about the version or type of the operation system, and do not rely on software vulnerabilities. In regard to the hardware, Nailgun is not restricted to any particular processor or SoC, and is able to work on various ARM-based platforms. Moreover, physical access to the platform is not required.

In the non-invasive debugging attack, we assume the `SPNIDEN` or `NIDEN` signal is en-

abled to attack the secure state or the non-secure state, respectively. We also make similar assumptions to the SPIDEN and DBGEN signals in the invasive debugging attack. We further assume the target platform is a multi-processor platform in the invasive debugging attack. Moreover, our attack requires access to the CoreSight components and debug registers, which are typically mapped to some physical memory regions in the system. Note that it normally requires non-secure EL1 privilege to map the CoreSight components and debug registers to the virtual memory address space.

5.4.2 Attack Scenarios

Inferring Encryption Key with Non-Invasive Debugging

The AES algorithm has been proved to be vulnerable to various attacks [122, 123, 153, 154, 156, 246]. The key vulnerability is the table-lookup based implementation, which is designed to improve the performance of AES, leaks the information about the encryption key. With the addresses of the accessed table entries, the attacker can efficiently rebuild the encryption key. In this attack, we assume there is a secure application running in TrustZone that holds the AES encryption key, and the secure application also provides an interface to the non-secure OS to encrypt a given plaintext. The non-secure OS cannot directly read the encryption key since TrustZone enforces the isolation between the secure and non-secure states. Our goal is to reveal the encryption key stored in the secure memory by calling the encryption interface from the non-secure OS.

The violation of privilege isolation described in Figure 16 enables a non-secure application to learn the information about the secure execution. Specifically, the ETM instruction trace aids to rebuild the addresses of the executed instructions while the ETM data-address

trace records the addresses of the data involved in data processing instructions (e.g., `ldr`, `str`, `mov`, and etc.). According to the access pattern of the AES, it is trivial to learn the instruction-address range that performs the table lookup and identify the memory addresses of the tables from the trace output, which further helps to retrieve the encryption key with the recorded data addresses. Note that the only information we require is the indices of the table entries accessed by the AES algorithm. Thus, to simplify the analysis and reduce the noise, we can use the address range filter in the ETM to trace only the address range that performs the table lookup.

To demonstrate the attack, we first build a bare-metal environment on an NXP i.MX53 Quick Start Board [182]. The board is integrated with a single Cortex-A8 processor that enables the data-address trace, and we build our environment based on an open-source project [283] that enables the switching and communication between the secure and non-secure states. Next, we transplant the AES encryption algorithm of the OpenSSL 1.0.2n [187] to the environment and make it run in the secure state with a predefined 128-bit key stored in the secure memory. A non-secure application can request a secure encryption with an `smc` instruction and a plaintext pointer in register `r0`.

Figure 18 demonstrates our attack process. We use a random 128-bit input as the plaintext of the encryption in ① and the corresponding ciphertext is recorded in ②. From the ETM trace stream, we decode the addresses of the accessed table entries in each encryption round and convert them into the indices of the entries by the base addresses of the tables, as shown in ③. With the indices and the ciphertext, it is trivial to reverse the AES encryption algorithm and calculate the round keys in ④. Finally, with the encryption key and accessed table entries in round 1, Nailgun decodes the original encryption key in ⑤.

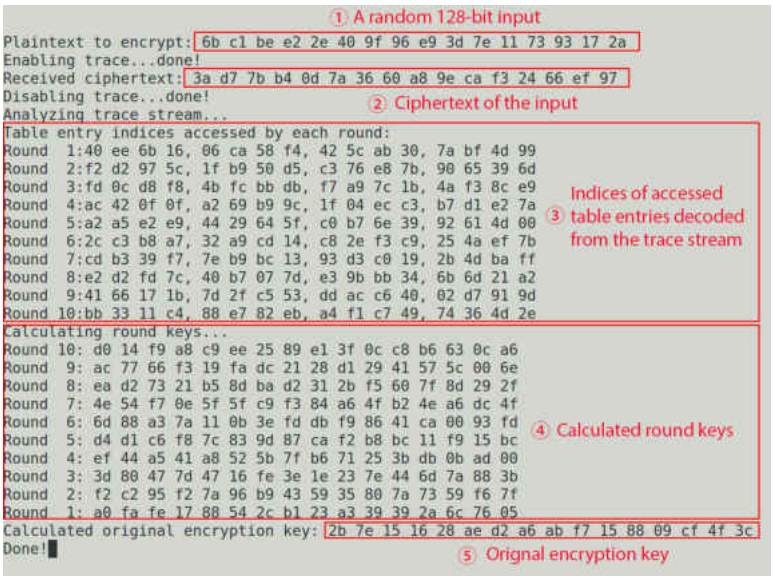


Figure 18: Retrieving the AES Encryption Key.

Note that previous side-channel attacks to the AES algorithm require hundreds of or even thousands of runs with different plaintexts to exhaust different possibilities. Nailgun is able to reveal the AES encryption key with a **single run** of an arbitrary plaintext.

Arbitrary Payload Execution with Invasive Debugging

The invasive debugging is more powerful than the non-invasive debugging since we can halt the target processor and access the restricted resources via the debugging architecture. Figure 17 shows a brief concept about the privilege escalation with invasive debugging, and we further expand the idea to achieve arbitrary payload execution.

The `EDITR` register offers an attacker the ability to execute instructions on the `TARGET` from the `HOST`. However, not all of the instructions can be executed via the `EDITR` register. For example, the execution of branch instructions (e.g., `b`, `bl`, and `blr` instructions) in `EDITR` leads to an unpredictable result. Meantime, a malicious payload in real world normally contains branch instructions. To bypass the restriction, Nailgun crafts a robust

approach to executing arbitrary payload in the high-privilege modes.

In general, we consider the execution of the malicious payload should satisfy three basic requirements: 1) Completeness. The payload should be executed in the non-debug state to overcome the instruction restriction of the EDITR register. 2) High Privilege. The payload should be executed with a privilege higher than the attacker owns. 3) Robust. The execution of the payload should not affect the execution of other programs.

To satisfy the first requirement, Nailgun has to manipulate the control flows of the non-debug state in the TARGET. For a processor in the debug state, the DLR_EL0 register holds the address of the first instruction to execute after exiting the debug state. Thus, an overwrite to this register can efficiently hijack the instruction control flow of the TARGET in the non-debug state.

The second requirement is tricky to satisfy. Note that the execution of the dcps instructions does not change the exception level of the non-debug state, which means that we need another privilege escalation in the non-debug state although the HOST can promote the privilege of the TARGET in the debug state. The smc instruction in the non-debug state asserts a Secure Monitor Call (SMC) exception which takes the processor to EL3, and we can leverage this instruction to enter EL3. However, we still need to redirect the execution to the payload after entering EL3. In each exception level, the incoming exceptions are handled by the handler specified in the corresponding exception vectors. In light of this, we manipulate the exception vector and redirect the corresponding exception handlers to the payload.

The third requirement is also critical since Nailgun actually modifies the instruction pointed by DLR_EL0 and the exception vectors indicated by the VBAR_EL3 registers. To avoid

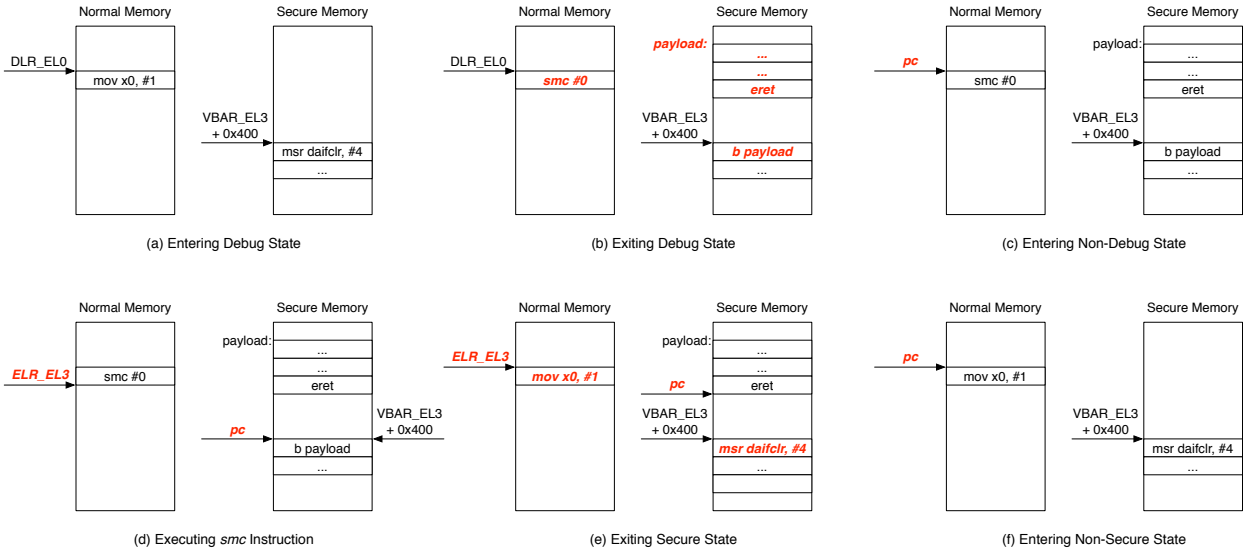


Figure 19: Executing Arbitrary Payload in the Secure State.

the side-effect introduced by the manipulation, Nailgun needs to rollback these changes in the TARGET after the execution of the payload. Moreover, Nailgun needs to store the value of stack pointers and general purpose registers at the very beginning of the payload and reverts them at the end of the payload.

We implement Nailgun on 64-bit ARMv8 Juno r1 board [28] to show that the *Implications 2-4* lead to arbitrary payload execution in EL3. The board includes two Cortex-A57 processors and four Cortex-A53 processors, and we use ARM Trusted Firmware (ATF) [30] and Linaro's deliverables on OpenEmbedded Linux for Juno [152] to build the software environment that enables both the secure and non-secure OSes. In the ATF implementation, the memory range 0xFF000000-0xFFDFFFFF is configured as the secure memory, and we demonstrate that we can copy arbitrary payload to the secure memory and execute it via an LKM in non-secure EL1.

Figure 19 describes the status and memory changes of the TARGET during the entire attack. The highlighted red in the figure implies the changed status and memory. In Fig-

ure 19(a), the TARGET is halted by the HOST before the execution of the mov instruction. Meantime, the VBAR_EL3 points to the EL3 exception vector. Since the SMC exception belongs to the synchronous exception and Juno board implements EL3 using 64-bit architecture, the corresponding exception handler is at offset 0x400 of the exception vector. Figure 19(b) shows the memory of the TARGET before exiting the debug state. Nailgun copies the payload to the secure memory and changes the instruction pointed by the DLR_EL0 to an smc instruction. Moreover, the first instruction in the 64-bit EL3 synchronous exception handler (pointed by VBAR_EL3 + 0x400) is changed to a branch instruction (the b instruction) targeting the copied payload. Then, the HOST resumes the TARGET, and the pc points to the malicious smc instruction, as shown in Figure 19(c). The execution of the smc instruction takes the TARGET to the status shown in Figure 19(d). Since the smc instruction is already executed, the value of the ELR_EL3 register is the address of the next instruction. Our manipulation of the exception handler leads to the execution of the payload, which can both perform malicious activities and restore the changed memory. At the end of the payload, an eret instruction is leveraged to switch back to the non-secure state. Figure 19(e) indicates the memory and status before the switch, and the changes to the non-secure memory and the EL3 exception vector is reverted. Moreover, the ELR_EL3 register is also manipulated to ensure the execution of the mov instruction. Finally, in Figure 19(f), the TARGET enters the non-secure state again, and the memory and status look the same as that in Figure 19(a).

Figure 20 shows an example of executing payload in TrustZone via an LKM. Our payload contains a minimized serial port driver so that Nailgun can send outputs to the serial port. To certify the attack has succeeded, we also extract the current exception level from

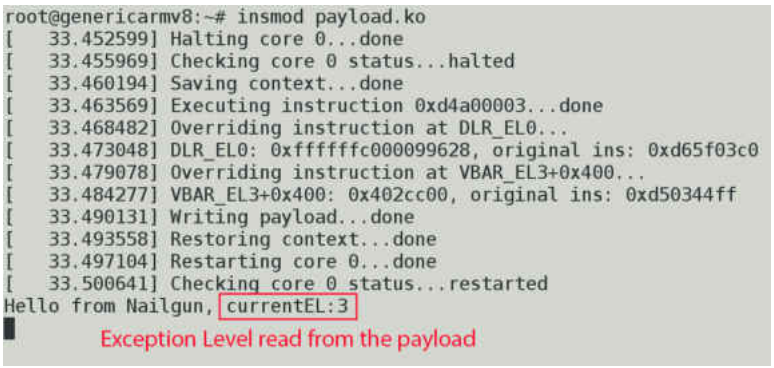


Figure 20: Executing Payload in TrustZone via an LKM.

the `CurrentEL` register. The last line of the outputs in Figure 20 indicates that Nailgun is able to execute arbitrary code in EL3, which owns the highest privilege over the whole system.

Fingerprint Extraction in a Real-world Mobile Phone

To learn the impact of Nailgun on the real-world devices, we also show that Nailgun is able to leak the sensitive information stored in the secure memory. Currently, one of the most used security features in the mobile phones is the fingerprint authentication [109, 167, 274], and the OEMs store the fingerprint image in TrustZone to enhance the security of the device [11, 102, 215]. In this experiment, we use Huawei Mate 7 [109] to demonstrate that the fingerprint image can be extracted by an LKM running in the non-secure EL1 with the help of Nailgun. The Huawei Mate 7 is powered by HiSilicon Kirin 925 SoC, which integrates a quad-core Cortex-A15 cluster and a quad-core Cortex-A7 cluster. The FPC1020 [88] fingerprint sensor is used in Mate 7 to capture the fingerprint image. This phone is selected since the product specification [89] and driver source code [272] of FPC1020 are publicly available, which reduces the engineering effort of implementing the attack.

As shown in the previous experiment, Nailgun offers a non-secure EL1 LKM the ability to read/write arbitrary secure/non-secure memory. To extract the fingerprint image, we need to know 1) where the image is stored and 2) the format of the image data.

To learn the location of the image, we decompile the TEE OS binary image, which is mapped to `/dev/block/mmcblk0p10`, and identify that a function named `fpc1020_fetch_image` is used to read the image from the fingerprint sensor. This function takes a pointer to an image buffer, an offset to the buffer, and the size of the image as parameters, and copies the fingerprint image fetched from the sensor to the image buffer. With further introspection, we find that Huawei uses a pre-allocated large buffer to store this image, and a pointer to the head of the buffer is stored in a fixed memory address `0x2efad510`. Similarly, the size of the image is stored at a fixed memory address `0x2ef7f414`. With the address and size, we extract the image data with Nailgun. Since the ARM architectures in Huawei Mate 7 and ARM Juno board are different, the implementations of Nailgun are also different.

The format of the image data is well-documented in the FPC1020 product specification [89]. According to the specification, each byte of the data indicates the gray scale level of a single pixel. Thus, with the extracted image data, it is trivial to craft a gray scale fingerprint image. Figure 21 shows the fingerprint image extracted from Huawei Mate 7 via Nailgun, and this result demonstrates that Nailgun is able to leak the sensitive data from the TEE in commercial mobile phones with some engineering efforts.

Nailgun in 32-bit ARMv8 and ARMv7 Architecture

In Section 5.2, we discussed the security implications of 64-bit ARMv8 debugging architecture, and similar implications exist in 32-bit ARMv8 and ARMv7 architecture. However,

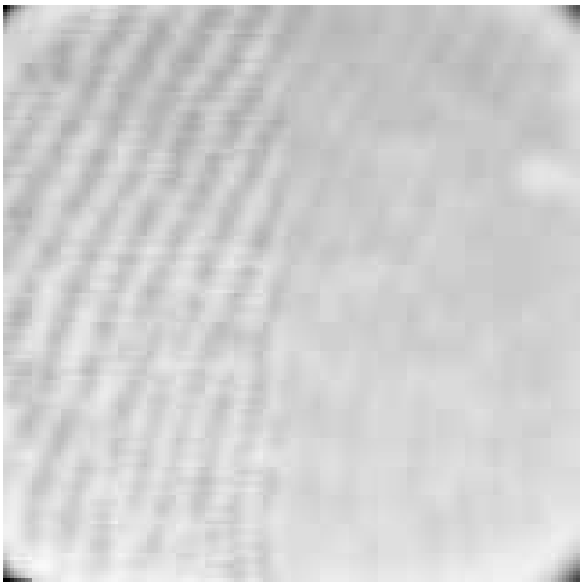


Figure 21: Fingerprint Image Leaked by Nailgun from Huawei Mate 7. Note that the right half of the image is blurred for privacy concerns.

there are also some major differences among the implementations of these architectures, and we discuss the differences in the following.

32-bit ARMv8 Debugging Architecture. We implement prototypes of Nailgun with 32-bit ARMv8 on Raspberry PI 3 Model B+ and Motorola E4 Plus. In this architecture, the steps of halting processor are similar to the aforementioned steps in 64-bit ARMv8 architecture, and the major difference between Nailgun on 32-bit and 64-bit ARMv8 architecture is the usage of the `EDITR`. In the 64-bit ARMv8, we directly write the binary representation of the instruction into the `EDITR`. However, the first half and last half of the instruction need to be reversed in the 32-bit ARMv8. For example, the binary representation of the `dcps3` instruction is `0xD4A00003` and `0xF78F8003` in 64-bit and 32-bit ARMv8, respectively. In the 64-bit ARMv8 architecture, we make the processor in the debug state execute this instruction via writing `0xD4A00003` to the `EDITR`. However, the instruction written to the `EDITR` should be `0x8003F78F` instead of `0xF78F8003` in the 32-bit ARMv8 architecture.

ARMv7 Debugging Architecture. In regard to ARMv7, we already implement Nailgun on Huawei Mate 7, and there are three major differences between Nailgun on ARMv7 and ARMv8 architectures. Firstly, the ECT is not required to halt and restart a processor in ARMv7. Writing `1` to the bit[0] and bit[1] of the Debug Run Control Register (DBGDRCR) can directly halt and restart a processor, respectively. Secondly, the `ITRen` bit of the `EDSCR` controls whether the `EDITR` is enabled in ARMv7 architecture. We need to enable the `ITRen` bit after entering the debug state and disable it again before exiting the debug state. Lastly, the `dcps` instructions are undefined in the ARMv7 architecture, and we need to change the `M` bits of the Current Program Status Register (CPSR) to promote the processor to the monitor mode to access the secure resource.

5.5 Countermeasure

5.5.1 Disabling the Signals?

Since Nailgun attack works only when the debug authentication signals are enabled, disabling these signals, in intuition, crafts an effective defense. However, according to the ARM Architecture Reference Manual [18, 20], the analysis results in Section 5.3, and the responses from the hardware vendors, we consider these signals cannot be simply disabled due to the following challenges:

Challenge 1: Existing tools rely on the debug authentication signals. The invasive and non-invasive debugging features are heavily used to build analysis systems [43, 65, 66, 69, 91, 142, 146, 160, 173, 282]. Disabling the debug authentication signals would directly make these systems fully or partially malfunction. In the ARMv7 architecture [18], the situation is even worse since the functionality of the widely used Performance Monitor Unit

(PMU) [3, 37, 74, 100, 173, 214, 285] also relies on the authentication signals. Since most of the aforementioned analysis systems attempt to perform malware detection/analysis, the risk of information leakage or privilege escalation by misusing the debugging features is dramatically increased (i.e., the debugging architecture is a double-edged sword in this case).

Challenge 2: The management mechanisms of the debug authentication signals are not publicly available. According to Section 5.3.3, the management mechanism of the debug authentication signals is unavailable to the public in most tested platforms. In our investigation, many SoC manufacturers keep the TRMs of the SoC confidential; and the publicly available TRMs of some other SoCs do not provide a complete management mechanism of these signals or confuse them with the JTAG debugging. The unavailable management mechanism makes it difficult to disable these signals by users. For example, developers use devices like Raspberry PI to build their own low-cost IoT solutions, and the default enabled authentication signals put their devices into the risk of being remotely attacked via Nailgun. However, they cannot disable these authentication signals due to the lack of available management mechanisms even they have noticed the risk.

Challenge 3: The one-time programmable feature prevents configuring the debug authentication signals. We also note that many of the tested platforms use the fuse to manage the authentication signals. On the one hand, the one-time programmable feature of the fuse prevents the malicious override to the debug authentication signals. However, on the other hand, users cannot disable these signals to avoid Nailgun due to the same one-time programmable feature on existing devices. Moreover, the fuse itself is proved to be vulnerable to hardware fault attacks by previous research [233].

Challenge 4: Hardware vendors have concerns about the cost and maintenance. The debug authentication signals are based on the hardware but not the software. Thus, without additional hardware support, the signals cannot be simply disabled by changing software configurations. According to the response from hardware vendors, deploying additional restrictions to the debug authentication signals increases the cost for the product lines. Moreover, disabling the debug authentication signals prohibits the legitimate debugging process such as repairing or bug fixing after a product recall, which introduces extra cost for the maintenance process.

5.5.2 Comprehensive Countermeasure

We consider Nailgun attack is caused by two reasons: 1) the debug authentication signals defined by ARM does not fully consider the scenario of inter-processor debugging, which leads to the security implications described in Section 5.2; 2) the configuration of debug authentication signals described in Section 5.3.2, which is related to the OEMs and cloud providers, and the management mechanism described in Section 5.3.3, which is related to the SoC manufacturers, make Nailgun attack feasible on real-world devices. Thus, the countermeasures discussed in this section mainly focus on the design, configuration, and management of the debug authentication signals. As a supplement, we also provide the defense that restricting the access to the debug registers, which may prevent the implementation of Nailgun. In general, we leverage the defense in depth concept and suggest a comprehensive defense across different roles in the ARM ecosystem.

Defense From ARM

Implementing additional restriction in the inter-processor debugging model. The key

issue that drives the existence of Nailgun is that the design of the debug mechanism and authentication signals does not fully consider the scenario of the newly involved inter-processor debugging model. Thus, redesign them and make them consider the differences between the traditional debugging mode and the inter-processor debugging model would keep the security implications away completely. Specifically, we suggest the `TARGET` checks the type of the `HOST` precisely. If the `HOST` is off-chip (the traditional debugging model), the existing design is good to work since the execution platforms of the `TARGET` and the `HOST` are separated (their privileges are not relevant). In regard to the on-chip `HOST` (the inter-processor debugging model), a more strict restriction should be required. For example, in the invasive debugging, the `TARGET` should check the privilege of the `HOST` and response to the debug request only if the `HOST` owns a higher or the same privilege as the `TARGET`. Similarly, the request of executing `dcps` instructions should also take the privilege of the `HOST` into consideration. The `HOST` should never be able to issue a `dcps` instruction that escalates the `TARGET` to an exception level higher than the current `HOST`'s exception level.

Refining the granularity of the debug authentication signals. Other than distinguishing the on-chip and off-chip `HOST`, we also suggest the granularity of the authentication signals should be improved. The `DBGEN` and `NIDEN` signals are designed to control the debugging functionality of the whole non-secure state, which offers a chance for the kernel-level (EL1) applications to exploit the hypervisor-level (EL2) execution. Thus, we suggest a subdivision to these signals.

Defense From SoC Manufacturers

Defining a proper restriction to the signal management procedure. Restricting the

management of these signals would be a reasonable defense from the perspective of the SoC manufacturers. Specifically, the privilege required to access the management unit of a debug authentication signal should follow the functionality of the signal to avoid the malicious override. For example, the management unit of the `SPNIDEN` and `SPIDEN` signals should be restricted to secure-access only. The restriction methods of current SoC designs are either too strict or too loose. On the ARM Juno SoC [28], all the debug authentication signals can only be managed in the secure state. Thus, if these signals are disabled, the non-secure kernel can never use the debugging features to debug the non-secure processor, even the kernel already owns a high privilege in the non-secure content. We consider this restriction method is too strict since it somehow restricts the legitimate usage of the debugging features. The design of the i.MX53 SoC [181], as opposed to ARM Juno SoC, shows a loose restriction. The debug authentication signals are designed to restrict the usage of the external debugger, however, the i.MX53 SoC allows an external debugger to enable the authentication signals. We consider this restriction method is too loose since it introduces a potential attack surface to these signals.

Applying hardware-assisted access control to the debug registers. Nailgun attack relies on the access to the debug registers, and the access is typically achieved by memory-mapped interfaces. Intuitively, the restriction to the access of these registers would help to enhance the security of the platform. However, we consider this restriction should be controlled in hardware-level instead of software-level. If the restriction is implemented by software running in the non-secure mode (e.g., the OS), the malware with kernel privilege may bypass it easily. If the restriction is implemented in the secure mode (e.g., TEE), it might introduce a significant performance overhead due to the semantic gap between the

two modes. In contrast, if the hardware-assisted access control applies, the access to the debug registers may be protected by hardware traps or interrupts. During the responsible disclosure to MediaTek, we learn that they have the hardware-based technology for TrustZone boundary division, and they are planning to use it to restrict the access to the debug registers to mitigate the reported attack.

Defense From OEMs and Cloud Providers

Keeping a balance between security and usability. With the signal management mechanism released by the SoC manufacturers, we suggest that OEMs and cloud providers disable all the debug authentication signals by default. This default configuration not only helps to protect the secure content from the non-secure state, but also avoids the privilege escalation among the non-secure exception levels. Meantime, they should allow the application with a corresponding privilege to enable these signals for legitimate debugging or maintenance purpose, and the usage of the signals should strictly follow the management mechanism designed by the SoC manufacturers. With this design, the legitimate usage of the debugging features from the privileged application is allowed while the misuse from the unprivileged application is forbidden. Moreover, since the debugging features are exploited via the CoreSight components and the debug registers, applying a similar restriction to the access of CoreSight components and debug registers can also form an effective defense.

Disabling the LKM in the Linux-based OSes. In most platforms, the debug registers work as an I/O device, and the attacker needs to manually map the physical address of the debug registers to virtual memory address space, which requires kernel privilege, to gain

access to these registers. In the Linux kernel, the regular approach to execute code with kernel privilege is to load an LKM. The LKMs in the traditional PC environment normally provide additional drivers or services. However, in the scenario of mobile devices and IoT devices, where the LKMs are not highly needed, we may disable the loading of the LKMs to prevent the arbitrary code execution in the kernel privilege. In this case, the attacker would not be able to map the debug registers into the memory even she has gained `root` privilege by tools like SuperSU [132]. Moreover, to prevent the attacker from replacing the stock kernel with a customized kernel that enables the LKM, the OEM may add some additional hash/checksums to verify the kernel image. Note that forbidding the customized kernel does not necessarily affect flashing a customized ROM, and the third-party ROM developers can still develop their ROMs based on the stock kernel.

CHAPTER 6 RESEARCH ON TRUSTED EXECUTION ENVIRONMENT

Recently, Trusted Execution Environments (TEEs) have been widely adopted in commodity systems for enhancing the security of software execution. This approach runs the security sensitive workloads in a trusted environment and all the running states of the workloads are guaranteed to be isolated from the potentially infected environment (e.g., the OS or hypervisor). The examples of TEE include but not limited to: Intel Software Guard eXtensions (SGX) [13, 107, 162], AMD Memory Encryption Technologies [136], ARM TrustZone Technology [31], x86 System Management Mode [116], AMD Platform Secure Processor [10], and Intel Management Engine (ME) [208].

We first introduce different TEEs in Section 6.1, and then analyze the security challenges of the TEEs in Section 6.2. A study of the deploying the TEEs on edge platform is presented in Section 6.3.

6.1 Trusted Execution Environments

In this section, we explain different Trusted Execution Environments. We category them into three types: 1) Ring 3 TEEs implemented via memory encryption; 2) ring -2 TEEs implemented via memory restriction; and 3) ring -3 TEEs implemented via co-processors. Next, we describe these three types of TEEs using the real world technologies.

6.1.1 Ring 3 TEEs via Memory Encryption

Intel Software Guard Extensions

Intel presented three introduction papers on Software Guard eXtensions (SGX) [13, 107, 162] in 2013. Intel SGX is a set of instructions and mechanisms for memory accesses added to Intel architecture processors. These extensions allow an application to

instantiate a protected container, referred to as an enclave. An enclave could be used as a TEE, which provides confidentiality and integrity even without trusting the BIOS, firmware, hypervisors, and OSes. Some of the researchers consider SGX as a new generation of TXT [70, 209]. Intel SGX is the latest iteration for trustworthy computing, and all future Intel processors will have this feature and use it as a TEE for addressing security problems. However, researchers raised security concerns about it. Recently, Costan and Devadas [70] published an extensive study on SGX. They analyzed the security features of SGX and raised concerns such as cache timing attacks and software side-channel attacks. Additionally, SGX tutorial slides from ISCA 2015 [117] mentioned that SGX does not protect against software side-channel attacks including using performance counters. Jain et al. [127] developed OpenSGX, an open-source platform that emulates Intel SGX hardware components at the instruction level by modifying QEMU.

AMD Memory Encryption Technologies

Recently, AMD introduced two new x86 features in ISCA 2016 and USENIX Security 2016 tutorials [135, 137]. One feature is called Secure Memory Encryption (SME), which defines a new approach for main memory encryption. The other is called Secure Encrypted Virtualization (SEV), which integrates with existing AMD-V virtualization architecture to support encrypted virtual machines. These features provide the ability to selectively encrypt some or all of system memory as well as the ability to run encrypted virtual machines, isolated from the hypervisor. AMD SME is a competitive technology with Intel SGX, and they provide ring 3 TEEs via memory encryption. Besides ring 3 TEEs, AMD memory encryption technologies can provide other system-level TEEs (e.g., hypervisor-level, ring -1).

The SEV technology can encrypt a virtual machine, and the OS running in the VM can be a TEE. AMD SME and SEV are upcoming coming technologies that will be supported in near future AMD chipsets.

6.1.2 Ring -2 TEEs via Memory Restriction

x86 System Management Mode and ARM TrustZone Technology create TEEs via *memory restriction*. Specifically, they use hardware (e.g., memory management unit) to setup access permissions of memory regions for the execution space, so the normal system software cannot access the execution space. Note that TEEs via memory restriction share the CPU with the normal system software in a time-slice fashion.

x86 System Management Mode

System Management Mode (SMM) [116] is a mode of execution similar to Real and Protected modes available on x86 platforms (Intel started to use SMM in its Pentium processors since the early 90s). It provides a hardware-assisted isolated execution environment for implementing platform-specific system control functions such as power management. It is initialized by the Basic Input/Output System (BIOS). SMM is triggered by asserting the System Management Interrupt (SMI) pin on the CPU. This pin can be asserted in a variety of ways, which include writing to a hardware port or generating Message Signaled Interrupts with a PCI device. Next, the CPU saves its state to a special region of memory called System Management RAM (SMRAM). Then, it atomically executes the SMI handler stored in SMRAM. SMRAM cannot be addressed by the other modes of execution. The requests for addresses in SMRAM are instead forwarded to video memory by default. This caveat, therefore, allows SMRAM to be used as a secure storage. The SMI handler is

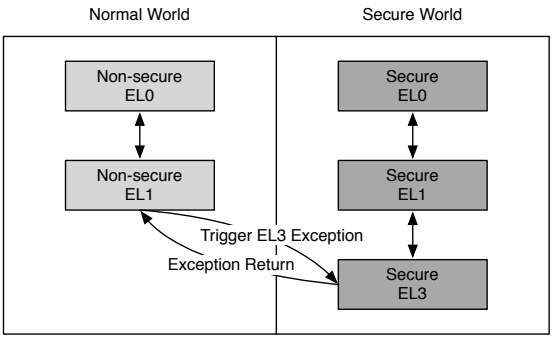


Figure 22: Processor Modes with ARM TrustZone.

loaded into SMRAM by the BIOS at boot time. The SMI handler has unrestricted access to the physical address space and can run privileged instructions (For this reason, SMM is often referred to as ring -2.) The `RSM` instruction forces the CPU to exit from SMM and resume execution in the previous mode.

ARM TrustZone Technology

ARM TrustZone technology [31] is a hardware feature that creates an isolated execution environment since ARMv6 around 2002 [19]. Similar to other hardware isolation technologies, it provides two environments or worlds. The Trust Execution Environment (TEE) is called the secure world, and the Rich Execution Environment (REE) is called the normal world. To ensure the complete isolation between the secure world and the normal world, TrustZone provides security extensions for hardware components including CPU, memory, and peripherals.

The CPU on a TrustZone-enabled ARM platform has two security modes: Secure mode and normal mode. Figure 22 shows the processor modes in a TrustZone-enabled ARM platform. Each processor mode has its own memory access region and privilege. The code running in the normal mode cannot access the memory in the secure mode, while the

program executed in the secure world can access the memory in normal mode. The secure and normal modes can be identified by reading the NS bit in the Secure Configuration Register (SCR), which can only be modified in the secure mode. As shown in Figure 22, ARM involves different Exception Levels (EL) to indicate different privileges in ARMv8 architecture, and lower EL owns lower privilege. The EL3, which is the highest EL, serves as a gatekeeper managing the switches between the normal mode and the secure mode. The normal mode can trigger an EL3 exception by calling a Secure Monitor Call (SMC) instruction or triggering secure interrupts, to switch to the secure mode, and the secure mode uses the Exception Return (ERET) instruction to switch back to the normal mode.

TrustZone uses Memory Management Unit mechanism to support virtual memory address spaces in both the secure and normal worlds. The same virtual address space in the two worlds is mapped to different physical regions. There are two types of hardware interrupts: Interrupt Request (IRQ) and Fast Interrupt Request (FIQ). Both of them can be configured as secure interrupt by configuring the IRQ bit and FIQ bit in SCR, respectively. The secure interrupt is directly routed to the secure EL3 ignoring the configuration of the normal world. ARM recommend that the IRQ is used as the interrupt source of the normal world and the FIQ is used as secure interrupt.

6.1.3 Ring -3 TEEs via Co-Processors

Intel Management Engine

The Intel Management Engine (ME) is a micro-computer embedded inside of all recent Intel processors, and it exists on Intel products including servers, workstations, desktops, tablets, and smart phones [208]. Intel introduced ME as an embedded processor in 2007.

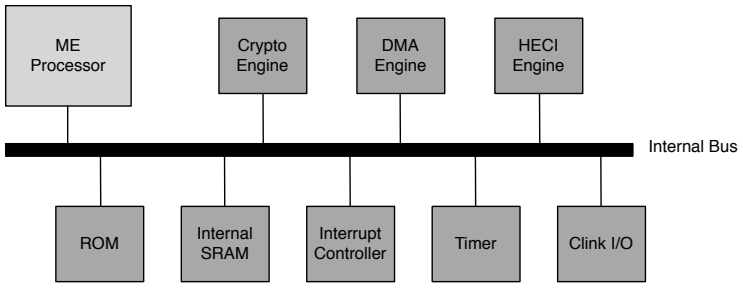


Figure 23: Architecture of Intel ME.

At that time, its main function was to support Intel Active Management Technology (AMT), and Intel AMT is the first application running in the ME. Recently, Intel started to use ME as a Trusted Execution Environmental (TEE) for executing security-sensitive applications. According to the latest ME book [208] written by an Intel Architect working on ME, a few security applications have been or will be implemented in ME including enhanced privacy identification, protected audio video path, identity protection technology, and boot guard.

Figure 23 shows the hardware architecture of ME. From the figure we can see that ME is like a computer; it contains a processor, cryptography engine, Direct Memory Access (DMA) engine, Host-Embedded Communication Interface (HECI) engine, Read-Only Memory (ROM), internal Static Random-Access Memory (SRAM), a timer, and other I/O devices. ME executes the instructions on the processor, and it has code and data caches to reduce the number of accesses to the internal SRAM. The internal SRAM is used to store the firmware code and runtime data. Besides the internal SRAM, ME also uses some Dynamic Random-Access Memory (DRAM) from the main system's memory (i.e., host memory). This DRAM serves a role as the disk; the memory pages of code/data that are not currently used by ME processor will be evicted from SRAM and swapped out to DRAM in the host memory. The region of DRAM is reserved by the BIOS when the system boots. This DRAM

is dedicated for ME use and the operating system cannot access it. However, the design of ME does not trust the BIOS and it assumes the host can access the reserved DRAM region.

AMD Platform Secure Processor

Although ME is for Intel processors, we can find similar technologies on AMD platforms. AMD Secure Processor [10] (also called Platform Security Processor or PSP) is a dedicated processor embedded inside of the main AMD CPU. It works with ARM TrustZone technology and ring -2 Trusted Execution Environments (TEE) to enable running third-party trusted applications. AMD Secure Processor is a hardware-based technology which enables secure boot up from BIOS level into the TEE. Trusted third-party applications are able to leverage industry-standard APIs to take advantage of the TEE's secure execution environment. Another example is System Management Unit (SMU) [159]. The SMU is a subcomponent of the Northbridge that is responsible for a variety of system and power management tasks during boot and runtime. The SMU contains a processor to assist [7]. Since AMD integrated Northbridge into the CPU, the SMU processor is an embedded processor inside of the CPU, which is same as Intel ME.

6.2 Challenges Towards Securing Hardware-assisted TEEs

6.2.1 Introduction

Although these well-designed and hardware-assisted TEEs provide secure execution environments, the code running in them could be buggy, which leads that the "trusted" execution environments (TEEs) are not trustworthy. While the argument is that the code base of a workload in a TEE is small enough so that the risk of having vulnerable code is low; however, due to the increasing complexity of the software and proliferation of using

TEEs in commodity systems, the developers keep increasing the size of the code in TEEs (e.g., OS running in TrustZone [30], hypervisor is deployed in SMM [40], Linux containers running in SGX [32]). The large code base of workloads in a TEE inevitably creates vulnerabilities that can be exploited by attackers. Even the security design and implementation of TEEs is flawless (e.g., perfect isolation and secure architectural design), we cannot prevent attacks that are due to the deployed buggy code. Even worse, the security features of TEEs might help the attackers. Leveraging these security features, attackers can implement higher level stealthy rootkits, which is extremely difficult to be detected by the existing defense tools. For example, the anti-virus tools running in the OS are not able to detect malicious code in an Enclave created by Intel SGX because the running memory in the Enclave is encrypted. SMM-based rootkits [155] have been used by National Security Agency as stealthy cyber weapons. Additionally, ring -3 rootkits [249] have been demonstrated by using Intel ME. Therefore, running the untrusted code in trusted execution environments raises a big security concern. Moreover, this can generate a series of research challenges since existing defense mechanisms can not be applied directly. The main objective of our research is to present this problem, discuss the research challenges, and provide potential directions to address them.

6.2.2 TEE-based Systems

TEE-based solutions are introduced in a variety of modern systems including cloud platforms (servers and clusters), endpoints (desktops and mobile devices), and edge nodes [225] (routers and gateways). In this section, we survey the applications and systems that leverage TEEs in x86 and ARM architectures.

SGX-based Systems and Attacks

Previous SGX-based systems such as Haven [46] ported system libraries and a library OS into an SGX enclave, which forms a large TCB. Arnautov et al. [32] proposed SCONE, a secure container mechanism for Docker that uses SGX to protect container processes from external attacks. Hunt et al. [111] developed Ryoan, an SGX- based distributed sandbox that enables users to keep their data secret in data-processing services. Schuster et al. [219] developed VC3, an SGX-based trusted execution environment to execute MapReduce computation in clouds. Karande et al. [138] secure the system logs with SGX. Shih et al. [227] leverages SGX to isolate the states of Network Function Virtualization (NFV) applications.

Schwarz et al. [220] attacks the SGX enclave via cache side channels, and demonstrates that the private key in the RSA implementation of `mbedTLS` can be extracted within five minutes. Other than RSA decryption, Ferdinand [53] also demonstrates a more efficient attack on the human genome indexing via SGX cache-based information leakage. AsyncShock [266] shows that the thread scheduling can be controlled by the attack, and the thread manipulation can be further used to exploit synchronization bugs inside SGX enclaves. SGX-Shield [221] provides secure address space layout randomization support for SGX programs. T-SGX [228] fight against the controlled-channel attack and ensures that the page fault will not be leaked.

SMM-based Systems and Attacks

In recent years, SMM-based research has appeared in the security literature. For instance, SMM can be used to check the integrity of higher level software (e.g., hypervisor

and OS). HyperGuard [210], HyperCheck [289], and HyperSentry [39] are integrity monitoring systems based on SMM. Moreover, National Science Foundation funded a project about using SMM for runtime Integrity checking last year [180]. SICE [40] presents a trusted execution environment for executing sensitive workloads via SMM on AMD platforms. SPECTRE [286] uses SMM to introspect the live memory of a system for malware detection. Another use of SMM is to reliably acquire system physical memory for forensic analysis [205, 263]. IOCheck [284, 288] secures the configurations and firmware of I/O devices at runtime. HRA [130] uses SMM for secure resource accounting in the cloud environment even when the hypervisor is compromised. MalT [285] progresses towards stealthy debugging by leveraging SMM to transparently debug software on bare metal. TrustLogin [287] protects user credentials especially passwords from theft in an untrusted environment. HOPS [145] uses SMM to create low-artifact process introspection techniques. As we can see that an array of SMM-based systems have been presented, and there is a need for us to develop novel techniques to secure the code of these systems.

Modern computers lock the SMRAM in the BIOS so that SMRAM is inaccessible from any other CPU modes after booting. Wojtczuk and Rutkowska demonstrated bypassing the SMRAM lock through memory reclaiming [210] or cache poisoning [269]. The memory reclaiming attack can be addressed by locking the remapping registers and Top of Low Usable DRAM (TOLUD) register. The cache poisoning attack forces the CPU to execute instructions from the cache instead of SMRAM by manipulating the Memory Type Range Register (MTRR). Duflot also independently discovered this architectural vulnerability [79], but it has been fixed by Intel adding SMRR [116]. Furthermore, Duflot et al. [78] listed some design issues of SMM, but they can be fixed by correct configurations in BIOS and careful

implementation of the SMI handler. Wojtczuk and Kallenberg [268] presented an SMM attack by manipulating UEFI boot script that allows attackers to bypass the SMM lock and modify the SMI handler with ring 0 privilege. The UEFI boot script is a data structure interpreted by UEFI firmware during S3 resume. When the boot script executes, system registers like BIOS_NTL (SPI flash write protection) or TSEG (SMM protection from DMA) are not set so that attackers can force an S3 sleep to take control of SMM. Butterworth et al. [56] demonstrated a buffer overflow vulnerability in the BIOS updating process in SMM, but this is not an architectural vulnerability and is specific to that particular BIOS version.

ME-based Systems and Attacks

Intel uses ME as a TEE to execute security sensitive operations [208]. In 2009, Tereshkin and Wojtczuk [249] demonstrated that they can implement ring -3 rootkits in ME by injecting the malicious code into the Intel Active Management Technology (AMT). DAGGER [240] bypasses the ME isolation using a similar technique in [249], but it hooks the ME firmware function `memset` because it is invoked more often. Skochinsky [232] discovers that the ME firmware on the SPI flash uses Huffman encoding to prevent reverse engineering for implementing rootkits. Recently, Intel disclosed an AMT vulnerability in ME (CVE-2017-5689 or INTEL-SA-00075 [120]). This bug allows attackers to remotely gain administrative control over Intel machines without entering a password [192], and this remote hacking flaw resides in Intel chips for seven years [92].

TrustZone-based Systems and Attacks

Mobile devices have been increased dramatically in past few years, security became

one of the major concerns of the users. ARM introduced TrustZone Technology and researchers used it to build an array of systems for enhancing the security of mobile devices. TrustDump [242] provides reliable memory acquisition by leveraging TrustZone. It uses a non-maskable secure interrupt to switch to the trust domain and introspects the memory of normal domain from trust domain. TZ-RKP [38] runs in the secure world and protects the normal OS kernel by event-driven monitoring. Sprobes [275] provides an instrumentation mechanism to introspect the normal OS from the secure world and guarantees the kernel code integrity. SeCReT [129] is a framework that enables a secure communication channel between the normal world and the secure world. TrustICE [243] provides a trusted and isolated computing environment for executing sensitive workloads. TrustOPT [241] presents a secure one-time password tokens by using ARM TrustZone technology on mobile devices. AdAttester [149] proposes a verifiable mobile ad framework that secures online mobile advertisement attestation using TrustZone. [52] suggest to use TrustZone to regulate the peripherals of devices (e.g., cameras) in restricted spaces. fTPM [201] is a firmware version of TPM 2.0 that implemented in ARM TrustZone. PrivateZone [128] uses TrustZone to create a private execution environment that is isolated from both the Rich Execution Environment and TEE. C-FLAT [4] fights against control-flow hijacking via runtime control-flow verification in TrustZone.

Qualcomm's use Secure Channel Manager (SCM) to interact with Qualcomm's Secure Execution Environment (QSEE) via SMC instruction, and [207] leverages this interface and exploits an integer overflow vulnerability to write arbitrary secure memory. Next, they rewrite the SMC handler table with this approach and gain arbitrary TrustZone code execution. [223] use `ret2user` to gain root privilege, and also a vulnerability of unchecked

bound to write one byte to almost any physical address, which finally leads to arbitrary payloads to be executed in TEE. ARMageddon [154] uses `Prime+Probe` cache attack to leak the information from secure world to normal world, and makes monitoring TrustZone code execution in normal world feasible.

6.2.3 Challenges and Directions

In this section, we detail the challenges for securing the hardware trusted execution environments. Moreover, we provide directions that might be able to address these challenges.

Hunting Bugs in TEE's code

The software running in a TEE might contain text-book vulnerabilities that can be easily exploited by attackers. Kallenberg and Kovah [133] found that "millions of BIOSes" are easy to be compromised because the known vulnerabilities of SMM code are never patched in the BIOS. Butterworth et al. [56] demonstrated a buffer overflow vulnerability in the BIOS updating process in SMM. Di [223] found vulnerabilities that are able to execute arbitrarily code in TrustZone code. Additionally, an array of SGX-based systems have been developed [32, 46, 111, 138, 219, 227], these SGX-based applications inevitably contain vulnerabilities due to their large code bases. There is a need for us to develop effective and efficient frameworks to find the vulnerabilities in the code/images running in the TEEs and reduce the chance of having vulnerable codes before attackers exploit it.

However, existing solutions of bug hunting can not be applied directly because the TEE's code requires particular environments (e.g., SMM and TrustZone) for execution. If we have the source code of the TEE software (e.g., SGX applications), we might be able

to modify existing static analysis or bug checking tools to identify bugs and minimize the number of vulnerabilities. However, if we do not have the source code but with a TEE's image, hunting bugs in binary images is very time-consuming and require heavy reverse-engineering efforts. Furthermore, other TEE's image such ME code might not be obtained due to hardware vendor's protection mechanism [232].

Therefore, there is a need to develop a framework to check the TEE's code before it runs in the high-privileged, isolated, and trusted environment. For instance, we can use symbolic execution (e.g., S2E platform) to analyze the SMI handler code and TrustZone firmware. Symbolic execution can help explore the execution paths of SMI handlers and TrustZone images, and discover the paths that lead to known exploitation. Since S2E can directly work on binaries on both x86 and ARM architectures, it can analyze commercial SMI handlers and TrustZone code without knowing the source code. Note that Bazhaniuk et al [183]. proposed using the similar way for analyzing SMI handler for the BIOS security. However, they only target on detecting the out-call functions (i.e., calling functions outside of the protected memory, SMRAM, that is controlled by attackers) in the SMI handler [211]. Moreover, not only targeting on SMI handler code on x86, we can apply the approach to the TrustZone firmware on ARM architecture. Furthermore, we can modify S2E plugins to work with other vulnerabilities such as buffer overflows and poisonous pointers [184] to help us validate the inputs from the untrusted environment.

Protecting Mechanisms within TEEs

It is impractical to have perfect code running in the TEEs, and the attackers will eventually find a vulnerability and exploit it at some point. However, existing TEEs lack of defense

mechanisms in the execution environments. For instance, the code running in SMM shares a single address space and paging is disabled [116]. Applications running in the SGX enclaves do not have basic protection mechanisms such as ASLR. In the normal computing environment (e.g., OS), we have an array of system-level defense mechanisms such as non-executable stack, data execution prevention, and address space randomization. However, these defense mechanisms are missing in the TEEs. Since we consider these environments are more secure than the normal OS, these basic system defense mechanisms are needed for securing the environment.

One of the defenses is to diversify TEE's environment. This increases the difficulty and cost for attackers to successfully exploit the bugs. With this approach, we can create dynamic TEEs. According to the Intel manual [116], system management mode has a very simple addressing mechanism. It disables the paging and works directly on physical addresses. When the system boots up, the BIOS initializes SMM and loads the SMI handler code to a physical memory address at SMM_Base + `0x8000`. SMM_Base represents the beginning of the SMRAM. Typically, the BIOS vendors set the SMM_base as `0xa0000` and this memory region overlaps with the VGA memory. We can randomize the base address of SMRAM for every boot or reboot by modifying the BIOS code. Specifically, we can randomly setup the SMRAM address in the BIOS for every reset signal. Note that the reset signal can be caused by a variety of power state changes including cold boot, warm boot, wake up from S3 (i.e., suspend to RAM), and so on. By randomizing the base address of SMRAM, it increases the difficulty for attackers to dump the SMRAM for exploitation or reverse engineering. Additionally, we can randomize the saved states in SMRAM, instead of at the fix location, SMM_base + `0xFC00`. Then, SMM attacks such as [184] would not

work since it requires to overwrite the SMM_base register at the save states area.

The execution environment of TrustZone is more complex than that of SMM. TrustZone has its own page tables and operates with memory management unit enabled. To diversify the execution environment of TrustZone, we can first randomize the location of the TrustZone firmware code. Within the TrustZone, it can support and run a Secure OS. We can implement the Address Space Layout Randomize (ASLR) technique on the Secure OS of the ARM trusted firmware. This addition can reduce the success rate of exploitation on attacks that leverage buffer overflows or return-oriented programming [222]. We may start this research direction with Coreboot [68] for the TEE like SMM, and Trusted Firmware [30] for the TrustZone.

Additionally, the randomization is needed in ring 3 TEEs (e.g., Intel SGX) as well. SGX-Shield [221] provides secure address space layout randomization support for SGX programs. Moreover, we can periodically or randomly instantiate an SGX enclave, and move the security sensitive workloads from one enclave to another. In this case, the associated states of the enclave are on the move so attacks depending on static information (e.g., memory addresses) might not work anymore.

Detecting a Compromised TEE

In practice, a TEE might be compromised due to the vulnerabilities in the code. However, detecting a compromised TEE is a very challenging problem because TEEs run at a high-privilege memory space that inaccessible from the system software (ring -2 TEEs) or use encrypted memory that their contents are mysterious without the key (ring 3 TEEs). For example, Intel SGX encrypts its code and data in enclaves; SMM and TrustZone code

is not accessible by the system software (e.g., OS). Because of these "security protection" features, a TEE can achieve a strong security guarantee. However, after compromising a TEE, attackers can implement undetectable advanced rootkits in it.

Embleton et al. [84] use SMM to implement a chipset-level keylogger and a network backdoor capable of directly interacting with the network card to send logged keystrokes to a remote machine via network packets. Schiffman and Kaplana [217] further demonstrated that with USB keyboards instead of PS/2 ones. Other SMM-based attacks focus on achieving stealthy rootkits [54, 155]. For instance, the National Security Agency (NSA) uses SMM to build an array of rootkits including DEITYBOUNCE for Dell and IRONCHEF for HP Proliant servers [155]. Several attacks [240, 249] have been demonstrated using ME to implement advanced stealthy rootkits. Tereshkin and Wojtczuk [249] injects malicious code into the Intel Active Management Technology (AMT) to implement ME ring -3 rootkits. DAGGER [240] is a DMA-based keylogger implemented in ME, and it captures keystrokes very early in the platform boot process, which enables DAGGER to capture harddisk encryption passwords. While proving a TEE as a strong isolated computing environment, having a method to detect a compromised TEE is a challenging task.

One potential approach to detect a compromised TEE is to use the performance implications, timing, or other side-channel information. For instance, we might be able to detect compromised SMM or TrustZone via the timing side-channel information. The intuition is that ring -2 TEEs share the main CPU with the system software in a time-slice fashion. This approach would not work for ring -3 TEEs since they run on separated processors, not the main CPU. Normally, SMM or TrustZone is invoked very few times or the execution times of them have some specific patterns for normal system operations. If we see a sys-

tem that dramatically changes its execution pattern (staying in SMM or TrustZone too long for sending out the sensitive memory pages via network packets) or invokes ring -2 TEEs very frequently (e.g., SMM-based keyloggers [84] generating SMIs for each keystroke), the system is more likely compromised. To detect the SMI invocation and its execution time, [262, 289] have implemented a tool called SMI Detector. The idea behind this tool is that the SMI invocations suspend all cores in the CPU, the SMI Detector can measure the missing time. A similar tool can work on TrustZone since it also shares the CPU in a time-slice fashion. Note that using the side-channels based approach for detecting compromised TEEs might not work for all the cases (e.g., timing side channel does not work for ring -3 TEEs). Other side channels including power consumption, cache access patterns, network traffic patterns can be considered for other cases.

Patching and Rejuvenation of TEEs

This subsection talks about the challenges on how to mitigate attackers from a compromised TEE and patch it to a good state. One simple approach is to use the system software to update the compromised TEE. However, if the TEE is compromised, it is likely that the system software is malicious, too. Thus the patching process running in the system software cannot be trusted. To ensure the restoring process is not tampered, we have to rely on a *Trust Base*. However, having such as a *Trust Base* is a challenging task.

For ring -2 TEEs, we might be able to use firmware as the *Trust Base*. This updating process works for some real world attacks such as Incursions (CERT VU#631788) [133]. In this attack, adversaries are able to bypass the isolation and get into SMM to run arbitrary code, the BIOS firmware is still protected by the `Write Enable` bit in the BIOS Control

register (BIOS_CNTL) [116]. As long as the attackers cannot flash the BIOS firmware, the system can perform a quick restart to destruct the SMRAM and re-initialize the compromised SMI handler. In this case, the update process of the compromised TEE from the firmware works. However, it is possible that attackers can bypass the write protection and reflash the firmware. For instance, Wojtczuk and Kallenberg [268] presented an SMM attack by manipulating UEFI boot script that allows attackers to bypass the BIOS write protection lock and modify the SMI handler with ring 0 privilege (CERT VU#976132). Moreover, Speed Racer [134] described a race condition that allows an attacker to subvert the firmware flash protection mechanism. In these attacking scenarios, how can we restore the SMI handler to a clean state if the firmware can not be trusted? If we assume the BIOS, SMM, and system software are all compromised, we need to rely on a component that does not have them in the Trusted Computing Base (TCB). One potential solution is the ring -3 TEEs such as Intel ME and AMD PSP. However, how to update ring -3 TEEs is another challenging task.

6.3 Preliminary Study of TEEs on Heterogeneous Edge Platforms

6.3.1 Introduction

The idea of Edge Computing [225, 226] suggests the deployment of additional edge nodes between the cloud server and the end-users, on which the latency-sensitive or privacy-sensitive computation is executed. Since the edge nodes are supposed to be as close as possible to the end-users, the latency is greatly reduced and the data privacy is improved to match the requirement of these computations. Meantime, those non sensitive computations are still on the cloud to take the advantage of cloud computing. Re-

cently, the usage scenarios [64, 99, 197, 271, 280, 291] and performance of Edge Computing [67, 292] are well studied by the researchers. However, less attentions are paid to the security and privacy of the Edge Computing, which puts the new-born Edge Computing infrastructure at risk.

Our research evaluate the performance of the TEEs to help analyze the feasibility of deploying them on heterogeneous edge platforms. Specifically, we first study the Intel SGX on a fog node following the Intel Fog Reference Design [118]. Since the infrastructure of Fog Computing is similar to the Edge Computing, we consider the fog node as a suitable candidate of the edge node. Meanwhile, the low-power consumption makes the ARM architecture to be a serious competitor to the x86 architecture on the edge platform. Thus, a study of ARM TrustZone technology on the ARM Juno development board [28] is presented in this paper. Finally, AMD processors are well-known by their low price, which is also a critical aspect in case of edge node due to its huge amount. Therefore, we also analyze the recent AMD Secure Encrypted Virtualization (SEV) technology for further comparison.

The results of our experiments show that the deployment of Intel SGX, ARM TrustZone technology, and AMD SEV introduces about 0.26%, 0.02%, and 4.14% performance overhead, respectively. Apparently, the overhead of the SGX and TrustZone technology are ignorable, and the overhead of SEV is reasonable due to the slowdown of a virtual machine.

6.3.2 Evaluation with Intel Fog Node

The Fog Node is introduced by Intel from the OpenFog Consortium [185], a consortium of high tech industry companies (e.g., Intel and Cisco) and academic institutions across the

world aimed at the standardization and promotion of fog computing in various capacities and fields. The processor on this node is 8-core Intel Xeon E3-1275 processor, which is a high-performance SGX-enabled processor. The 32GB DDR4 memory also meets the requirement of usage scenario of fog computing. In regard to the software, we leverage the open source Tianocore BIOS and 64-bit Ubuntu 16.04 to setup the node. Due to the similarity of the Fog Computing and Edge Computing, we consider this machine also matches the design of Edge Computing and can be directly used as an edge node. Therefore, we use the fog node to simulate the edge node in the performance analysis of Intel SGX.

In this section, we create applications based on the SGX SDK 1.9 [119], and use them to conduct the experiments to measure the performance overhead. Specifically, we evaluate the time consumption of the context switch in SGX, the performance slowdown of transplanting the computation into enclave, and the slowdown of the overall system when SGX is involved, respectively.

Context Switch

Regarding to the experiments in this section, we use an empty `ECALL` function to achieve the context switch. Once the function is called, the CPU will switch to the enclave mode, while the exit of the function implies the exit of the enclave mode. To measure the time consumption, the `RDTSC` instruction is used to read the elapsed CPU cycles. Note that the execution of this instruction is forbidden in enclave mode, so we cannot measure the required time to entering or quitting the enclave mode separately. Instead, we calculate the time consumption of a complete context switch cycle (i.e., enter and then quit the enclave mode). Moreover, the parameter transferring between the enclave mode and normal mode

Table 14: Context Switching Time of Intel SGX on the Fog Node (μs).

Buffer Size	Mean	STD	95% CI
0 KB	2.039	0.066	[2.035, 2.044]
1 KB	2.109	0.032	[2.107, 2.111]
4 KB	2.251	0.059	[2.247, 2.254]
8 KB	2.362	0.055	[2.359, 2.366]
16 KB	2.714	0.036	[2.712, 2.716]

depends on an additional buffer, and the size of the buffer affects the efficiency of the context switch. Therefore, we use different buffer sizes to conduct the evaluation. To reduce the nondeterminacy of the experiments, we configure the CPU frequency to be a fixed value (4GHz) and repeat the experiment for $1,000$ times.

Table 14 shows the context switching time of Intel SGX on the Intel Fog Node. If no parameter is required, the context switch requires 2.039 μs, this is the approximate time consumption for the CPU mode switching. However, in most usage scenarios, the parameter transferring is required. The time consumptions come to 2.109 μs, 2.251 μs, 2.362 μs, and 2.714 μs when the sizes of the parameters are 1KB, 4KB, 8KB, and 16 KB, respectively.

Sensitive Computation

Since the TEEs are used to secure the sensitive computation, we are eager to know the overhead of moving the sensitive computation into the TEEs. In this experiment, we use an open-source MD5 implementation [206] following the RFC 1321 standard to simulate the sensitive computation, and measure the time consumption of calculating the MD5 inside and outside the enclave mode. Without loss of generality, we use a pre-generated random string with $1,024$ characters as the target of the MD5.

Table 15: Time Consumption of MD5 (μs).

CPU Mode	Mean	STD	95% CI
Normal	4.734	0.095	[4.728, 4.740]
Enclave	6.737	0.081	[6.732, 6.742]

As shown in Table 15, the MD5 calculation requires 4.734 μs in normal mode and 6.737 μs in enclave mode. We note that the calculation in the enclave mode requires about 2.003 more microseconds than the calculation in the normal mode, and this difference is close to the context switching time. This result shows that the CPU performance in normal mode and enclave mode are similar, and the overhead of moving the sensitive computation to the TEEs depends on the overhead of the context switch.

Overall Performance

While keeping the sensitive computation running inside the TEE, we also want to make sure that the performance of the non-sensitive computation on the edge node would not be affected. To simulate the frequent sensitive computation on the edge node, we switch to the enclave mode every one second and calculate the MD5 of a 1024-length string. A dedicated CPU benchmark, GeekBench [195], is used to measure the performance of the CPUs. To avoid the unpredicted affects from the other software in the system, we make the sensitive computation and the benchmark to be executed in the same core. The single-core performance score with and without the sensitive computation are compared to learn the overall performance overhead. The experiment is repeated for 100 times to reduce the test errors.

Table 16 shows the performance score given by GeekBench. The single-core performance scores with and without secure computation are $4,327.33$ and $4,306.46$, respec-

Table 16: Performance Score by GeekBench.

Sensitive Computation	Mean	STD	95% CI
No	4327.33	17.124	[4323.974, 4330.686]
Yes	4306.46	14.850	[4303.550, 4309.371]

tively, and the performance slowdown is 0.48%. Apparently, the performance overhead of the computation inside the SGX enclave is ignorable even we switch to the enclave mode every one second.

6.3.3 Evaluation with ARM Juno Board

The ARM Juno Board [28] is an official software development platform for ARMv8 architecture [20], and it represents the most recent hardware design of ARM. We consider the further ARM-based edge node will follow this design and thus perform our experiments on the Juno board. The Juno r1 development board contains a dual-core Cortex-A57 cluster and a quad-core Cortex-A53 cluster, and all the processors in the clusters are equipped with ARM TrustZone technology. The main memory of the board is an 8GB DRAM. We also use the ARM Trusted Firmware (ATF) [30] to enable the firmware support for TrustZone. The Android deliverable image for Juno board provided by Linaro [152] is used to be the operating system of the non-secure mode.

Similar to the experiments running with the Intel SGX, we evaluate the performance overhead of the context switch, sensitive computation, and the overall system, respectively.

Context Switch

The `SMC` instruction is frequently used to achieve the switch between the secure mode and non-secure mode in many TrustZone-related systems. Thus, we also use this instruction to trigger the switch. To accurately evaluate the time consumption, we leverage the

Table 17: Context Switching Time of ARM TrustZone (μs).

Step	Mean	STD	95% CI
Non-secure to Secure	0.135	0.001	[0.135, 0.135]
Secure to Non-secure	0.082	0.003	[0.082, 0.083]
Overall	0.218	0.005	[0.218, 0.219]

Performance Monitor Unit (PMU) [20] to record the elapsed CPU cycles. Since the PMU can be used in both the secure and non-secure mode, we can learn the time consumption of the switching from non-secure mode to secure mode as well as that of the switching from secure mode to non-secure mode. Unlike the SGX, the parameters transferring in TrustZone is achieved by sharing the general purpose registers instead of using buffers. Therefore, the parameters involve no additional overhead. In the experiments, we configure the CPU to run at 1.15GHz and repeat the context switch for $1,000$ times.

Table 17 shows the context switching time of secure and non-secure mode. The switch from non-secure mode to secure mode requires 0.135 μs while the switch from secure to non-secure mode requires 0.082 μs, and the overall switching time is 0.218 μs. The small standard deviations also show that the time consumption of the context switch is stable.

Sensitive Computation

In this section, we integrate the aforementioned MD5 implementation to both a kernel module and the ATF. In the kernel module, we measure the time consumption of directly using the MD5 implementation and using the `SMC` instruction to invoke the MD5 implementation inside the ATF. The other setups of the experiments are similar to the experiments with the Intel SGX.

The result in Table 18 shows that it takes 8.229 μs to calculate the MD5 in the non-

Table 18: Time Consumption of MD5 (μs).

CPU Mode	Mean	STD	95% CI
Non-secure	8.229	0.231	[8.215, 8.244]
Secure	9.670	0.171	[9.660, 9.681]

Table 19: Performance Score by GeekBench.

Sensitive Computation	Mean	STD	95% CI
No	984.70	1.878	[984.332, 985.068]
Yes	983.44	3.273	[982.799, 984.082]

secure kernel module while the computation in the secure mode takes 9.670 μs. The increased computation time is 1.441 μs, which is much larger than the context switch discussed above (0.218 μs). Thus, we consider that the CPU performance is decreased in the secure mode.

Overall Performance

Similar to the experiments on SGX, we use an application to simulate the frequent sensitive computation and leverage the GeekBench 4 application [196] from Google Play Store to measure the CPU performance. The benchmark is executed for 100 times to reduce test errors.

From the Table 19, we find that the single-core performance score decreases from 984.70 to 983.44 when the sensitive computation is involved. The decrease percentages is 0.13%, which is ignorable. Therefore, we consider the slowdowns would not affect the performance of the edge nodes.

6.3.4 Evaluation with AMD EPYC CPU

To study the performance overhead of the AMD SEV, we use a machine with an AMD EPYC-7251 CPU [9], which contains 8 physical cores and 16 logic threads. As to the software, the operating system we use is Ubuntu 16.04.5 LTS with a customized SEV-enabled Linux kernel 4.15.10. The hypervisor we use is KVM 2.5.0.

Context Switch

In the SEV-ES architecture, VMEXIT events are splitted into two types, Automatic Exits (AE) and Non-Automatic Exits (NAE). In the system where SEV-ES is enabled, only AE can successfully trigger the VMEXIT event, which will cause a full world switch and the control will be transferred back to the hypervisor. During this process, the CPU hardware will save and encrypt all guest register states before loading the hypervisor.

To create an AE, we chose VMMCALL instruction. Though other instruction exists, the KVM we use currently does not support them. VMMCALL is meant as a way for a guest to explicitly call the hypervisor, and no Current Privilege Level (CPL) checks will be performed, thus the hypervisor can decide whether to make this instruction legal at the user-level or not, which also means we can add function by hooking the VMMCALL handler [8].

Since we can know the total switch time by sending an empty VMMCALL instruction, which is also the real thing what we are interested in, we did not record the time consumption of vmexit or vmentry event but record the total time consumption instead. From our experiment, we find that the average switch overhead is 3.09 μs, and this is because a vmexit event is triggered every time, and the CPU has to save and encrypt the guest

Table 20: Time Consumption of MD5 (μs).

CPU Mode	Mean	STD	95% CI
Guest OS	3.66	0.126	[3.602, 3.720]
Host OS	0.70	0.005	[0.697, 0.702]

state before switching to the hypervisor mode to protect guest data. Meantime, when CPU returns to Guest mode, it has to load and decrypt guest state.

Sensitive Computation

To evaluate the performance overhead of the sensitive computation, we study the time consumption of running sensitive computation software in both host and guest OS respectively. The each experiment is executed $1,000$ times. We restart the host operating system to make sure there is no other factor to impact our result. In the Host OS, we simply run MD5 and measure the time. To better simulate the real SEV executing environment, we call VMMCALL instruction every time the MD5 finishes to trigger the guest-hypervisor switch.

From Table 20 we can see that executing MD5 in Guest OS takes almost the same amount of time with running MD5 in the Host OS. Since we do not send any command with VMMCALL, the hypervisor does not have to do any extra calculation. Thus, we can see that the computation running in an SEV-enable guest does not introduce extra overhead compared to running in the Host OS.

Overall Performance

We use GeekBench 4 to evaluate the influence of frequent sensitive computation running in the SEV-ES enabled guest to the host. To simulate this, we run MD5 in Guest OS

Table 21: Performance Score by GeekBench.

Sensitive Computation	Mean	STD	95% CI
No	3425.05	41.016	[3417.011, 3433.089]
Yes	3283.15	32.772	[3276.727, 3289.573]

every 1 second and VMMCALL instruction is sent every time after MD5 hash finishes. By comparing the performances of with and without running sensitive computation in Guest OS, we can learn the overall extra overhead. We execute benchmark for 100 times.

From the Table 21, we can see that the performance score drops from 3425.05 to 3283.15 in average, and the decrease percentage is about 4.14%. Comparing with the experiments on Intel SGX and ARM TrustZone technology, we consider the AMD involves a higher performance overhead due to the heavily context switch between the hypervisor and guest OS.

CHAPTER 7 RESEARCH ON TRAFFIC SIGNAL INFRASTRUCTURE

7.1 Introduction

Traffic signal systems maintain the safety and coordination of all vehicles and pedestrians traversing public roads. Historically, these systems have proven themselves worthy. Governed by the Institute of Transportation Engineers (ITE) [124], the group has looked to excel in safety, education, and standardization of vehicle traffic intersections. The work of ITE has led to the general population trusting these systems and has delivered the expectation that a trip by vehicle or foot will be a safe journey.

Early implementations of traffic signal systems were based upon electro-mechanical controls. In the electro-mechanical systems of yesterday, the devices used nothing more than rotating gears and wheels that would spin and align contact leads to pass electricity to light bulbs contained in a traffic signal system [253]. Simple enough, these devices worked but lacked any technology to provide real-time reconfiguration to allow for changes to accommodate ever-changing vehicle traffic flows.

Fast-forward some years, modern traffic signal systems have ushered in numerous technologies due to advancements in computing and the modern need for more efficient systems. With emerging smart cities [254], the new version of traffic signal systems have ushered in numerous advancements compared to the systems of yesterday. Featuring improvements such as Linux based operating systems and network architectures spanning hundreds of miles, intelligent transportation control systems have achieved a degree of efficient control over vehicle traffic that has long been sought after.

With new advancements that have been developed and deployed, it is critical that

traffic signal systems be proven for safety and security above all. Previous security research of the traffic signal systems [60, 63, 62, 144, 150] mainly focus on the security of the traffic controllers and the wireless network deployed in the traffic signal system, and show that existing traffic systems are vulnerable. However, the security of the other parts (e.g., the Malfunction Management Unit and Cabinet Monitor Unit) in the traffic signal system are left out.

In this chapter, we share our pathway and execution for finding and exploiting flaws found in traffic signal systems (e.g., specified by NEMA TS-2 [170] and ITS [115] Cabinet standards). Our work does not focus on a specific component, but instead analyze the security of the whole traffic signal system. Our analysis results show that an array of attacks can be easily launched by adversaries against these systems such as bypassing access controls, disabling monitoring alerts/units, manipulating traffic patterns, or causing denial of services. Moreover, we show that attackers can perform an all-direction greens attack against vehicle traffic signal systems. To the best of our knowledge, it is the first time that such a severe attack has been demonstrated. By setting up a standard traffic signal system locally in our laboratory and leveraging a traffic signal system laboratory in a municipality, we test and verify the effectiveness of all the presented attacks on typical traffic signal systems following the TS-2 and ITS standards. Furthermore, we provide our security recommendations and suggestions for the vulnerabilities and attacks we confirmed.

7.2 Related Work

Previous work [62] investigated the security of vehicle traffic signal systems. In their analysis, the researchers identified vulnerabilities about the deployed wireless network and

operating system of the traffic controller. Exploiting these vulnerabilities, the researchers were able to control intersections on-demand to give them the ability to completely manipulate vehicle traffic progression. Cerrudo [60] presented vulnerabilities on the wireless sensors of the vehicle traffic signal systems. These vulnerabilities allow attackers to take complete control of the devices and send fake data to vehicle traffic signal systems. By leveraging these flaws, adversaries can cause traffic jams in a city. Laszka *et al*. [144] developed a method for evaluating the transportation network vulnerability, and their method is tested on randomly generated and real networks. Their approach can further identify critical signals that affect the congestion. Li *et al*. [150] presented risk-based frameworks for evaluating the compromised traffic signals and provided recommendations for the deployment of defensive measures in the vehicle traffic signal systems. [63] focuses on the vulnerability of the I-SIG system and shows that traffic congestion could be introduced by data spoofing attack from even a single attack vehicle. Unlike these work, we target the ATCs featuring two standards (i.e., ITS and TS-2) and advance their work in the following aspects: 1) We analyze the security of the entire traffic signal system in both ITS and TS-2 standards and summarize the security implications; 2) we show that stealthy manipulation to the traffic signal system is feasible via a diversionary cabinet access tactic; 3) we demonstrate the feasibility of the all-direction greens attack via bypassing the MMU/CMU.

7.3 Attack Surface Analysis

In this section, we analyze the security of existing vehicle traffic signal systems and summarize potential security implications. Note that the summarized implications are based on the study in the partnering municipality, and they may also apply to other mu-

nicipalities using the same device.

7.3.1 Access to the Traffic Signal System

When breaching the perimeter to access traffic signal systems, an attacker will encounter both physical access and/or remote access restrictions. In the case of a network intrusion, an attacker will likely gain access to more than one ATC due to the uniform use of network restriction mechanisms. With a physical intrusion, an attacker would first need to breach a traffic signal cabinet or operation center, then proceed to escalate privileges through a regional transportation network. In this section, both access methods are discussed to provide a through pathway to regional traffic signal access.

Physical Access

As mentioned in Section 2.4, the hardware devices in the traffic signal system are normally placed in a roadside cabinet. To avoid unauthorized access or destruction, the cabinet is protected by a Corbin #2 lock and key. This key is held by technicians who maintain the technology inside the cabinet. To assist with physical monitoring, surveillance cameras may be deployed to monitor potential access to the traffic cabinet.

Cabinet Keys. According to the cabinet specifications [115, 170], both the ITS and TS-2 cabinets shall be provided with a Corbin #2 Type key. Due to the large amount of deployed cabinets under these standards, we looked to verify this within our testing municipality. Through inquiry and testing, we verified that all of our testing municipalities traffic signal cabinets can be opened with the default Corbin #2 key.

With further research, we found that the Corbin #2 master key is sold online. For the price of $5 USD, the key is marked with the ability to open most traffic signal cabinets in

the United States. Upon further examination, the purchased key was proven to be an exact match to the cabinets that are used by our partnering municipality and standards that we are investigating. This key would allow us to open all traffic signal cabinets deployed by the municipality.

Implication 1: A large number of traffic signal cabinets can be opened with a Corbin #2 key purchased online..

Surveillance Cameras. Prior research has commented on the difficulty of beating surveillance cameras when gaining physical access to traffic cabinets [62]. However, our analysis shows a different result. In the municipality we investigated, there are 750 vehicle intersections. According to the municipality officials, only 275 vehicle intersections are covered by traffic cameras, which leaves more than 60% intersections of the traffic network unsurveilled. Without a surveillance camera, physical access to the traffic cabinets would be undetectable.

Implication 2: Physical access to the traffic signal cabinets is out of watch of surveillance cameras in more than 60% intersections of the investigated municipality..

Door Status Monitoring. In the ITS cabinets, the status of the door can be monitored by the CMU [115]. Specifically, the ATC sends a Type 61 query command [115] to the CMU, and then the current status of the cabinet door is returned in the 31st byte of the response. In real-world deployments, we learn that the Model 2070 ATC [121], which is deployed in the investigated municipality, writes the door alarm message to log file, then after some time, the log file is forwarded to the parties who are monitoring the system. However, we are informed by our test municipality that the forwarding of the log files is kept to a

low frequency (typically every one-to-five minutes) to reduce network congestion. This one-to-five minute gap offers a perpetrator a chance to clean up the log files before they get forwarded through the Model 2070's user interface. According to the test municipality, none of the cabinet door alarms are currently being monitored across the 750 vehicle intersections that they encompass.

Implication 3: The door status of traffic signal cabinets may not be monitored in real-time or at all. Alarms may be cleared from the system by an attacker..

Remote Access

As shown in previous work [62], a number of transportation systems use the insecure IEEE 802.11 wireless access points for network communications. The insecure wireless network would allow a perpetrator to remotely connect to a traffic network and access networked hardware inside.

While the network of some traffic signal systems is isolated from the Internet for security concerns, we do find that the public IP addresses of traffic signal systems are publicly accessible. The Shodan [229] website provides a search engine for internet-connected devices where reports can be generated containing IP addresses and signatures of devices meeting search criteria. With keywords such as `NTCIP` or `Econolite`, we are able to identify the IP address of a number of ATCs. Note that the keyword `Econolite` is traffic signal system manufacture who makes ATCs for ITS cabinets.

Additionally, due to the engineering efforts required for system updates, the Linux kernel in the ATCs is normally out-dated and is vulnerable to multiple existing attacks [71]. The ATCs used at our partnering municipality were confirmed to running the Linux 2.6.39

kernel network wide. Moreover, since the SSH/FTP connection is required in the ATCs [2], a perpetrator may also leverage known attacks [252] to gain access to the system. During our analysis, we found that both the deployed Intelight Model 2070 ATCs and Siemens Model 60 ATCs use default credentials for the SSH and Telnet connections. According to our partnering municipality, they were not aware of the ability to login to the ATC over SSH. This poses an interesting predicament as it appears that there may be additional municipalities that may not have an understanding that they are vulnerable to network attacks conducted via SSH.

Implication 4: SSH connections to ATCs are possible via the publicly exposed IP addresses and default credentials..

7.3.2 Traffic Signal Control

As described in Section 2.4, the ATC is used to configure traffic signal patterns and timing. In the devices we investigated, the Intelight Model 2070 ATC uses D4 software [90] for configuration while the SEPAC software [231] is used in the Siemens Model 60 ATC. Since the ATCs follow the same standard [2], the basic functionalities of the different software are the same.

With Physical Access

To reduce the complexity of using the software, the ATCs are equipped with a series of control buttons on the front panel. With the buttons and configuration menus, one can easily specify the configurations of the ATC including different traffic signal patterns, the internal clock, and the status of MMU/CMU.

In our investigation, we found out that the configuration of the ATC does not require

authentication. In other words, it requires no credentials to access the front control panel of the ATC, that can be used to configure the ATC freely. While access codes can be set to control access to this front panel, our partnering municipality did not do so. Therefore, once physical access is gained to the ATC, a perpetrator may modify the configuration of the ATC without any restrictions.

With Remote Access

In the ATC system, the D4 and SEPAC work as traffic control software in the Linux system. Naively, an attacker can gain remote access to the front panel controls by connecting into the Linux subsystem of controller. With the D4 software, an attacker that launches a connection will be displayed a remote terminal with the same controls that are offered on the Model 2070 front panel. With the SEPAC software an attacker can gain access to the front panel of by launching the front panel binary contained in the `/opt/sepac/` directory.

With remote access to the ATC via SSH, one can also control the traffic signals following the specification described in [2]. Specifically, the ATC is provided with seven serial communication ports, which are mapped as devices in the Linux `/dev` directory. According to the specification, Serial Port 3 (`/dev/sp3s`) and Serial Port 5 (`/dev/sp5s`) are used for in-cabinet device communications. Thus, directly writing a `Type 0` [170] command frame to the Load Switch relays achieves control of the traffic signal. To avoid conflict with the D4/SEPAC software, an attacker can stop the control software and their actions.

Similar to the aforementioned configuration with physical access, writing commands to the serial ports does not require any authentication in the investigated devices.

Implication 5: The configuration of ATCs and the communication between the ATC and the traffic control signal do not require any authentication..

7.3.3 Conflict Status Control

Recall that the MMU/CMU is in charge of detecting the conflict between the ATC configuration and predefined forbidden patterns. The forbidden patterns in the MMU and CMU are specified by the Programming Card and Datakey, respectively. Thus, to control the conflict status, a perpetrator needs to override the configuration in the Programming Card or Datakey.

MMU Programming Card

The conflict status on the MMU is defined by the compatibility between channels on the Programming Card [170]. Configuration is accomplished through the use of soldered wire jumpers. Therefore, to override the configuration, the perpetrator needs to resolder the wire jumpers to specify the required status.

CMU Datakeys

According to the cabinet specification [115], the CMUs in ITS cabinets use the LCK4000 Datakey [35]. We find that the LCK4000 is a 4 KB serial EEPROM memory chip molded into a plastic form-factor resembling a house key. Designed and manufactured by ATEK Access Technologies [36], the Datakey serves as an unencrypted configuration storage unit for the CMU that includes the known safe-states for an intersection housed in a defined byte-array [115]. Located on the ATEK Access Technologies website, we find that the company offers memory flashing devices for the LCK4000, and also instructions for making your own reader and writer based upon the Microwire serial communication protocol [34].

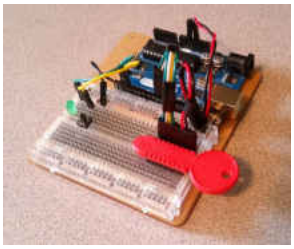


Figure 24: Datakey LCK4000 Microwire Flasher built upon the Arduino Platform. The red key is the LCK4000 Datakey.

To configure the Datakey, we would be able to buy an EEPROM memory flashing unit directly from ATEK. However, to learn the bar of overriding the configuration, we built a customized Datakey access tool by using an Arduino Uno starter-kit [17]. Following the Microwire serial protocol specification [34] found on ATEK's website, we are able to construct our own flashing device as shown in Figure 24. Similar to the control of traffic signals, the configuration of the Datakey requires no authentication, and our simple device would allow us to read and write configurations on-demand without any restriction.

Implication 6: The configuration of the conflict status control does not require any authentication..

7.3.4 Troubleshooting of the Traffic Signal System

Wireless 802.11 deployments in traffic networks are generally linear in communication flows. That is, due to the geography that must be covered in these networks, the use of redundant protocols such as spanning-tree is not seen due to the extra cost needed to design and install additional equipment. If there are no redundant loops in the network architecture, one can easily disable network communications across a linear communication chain by disabling an upstream communication node (i.e., *an intersection*). Thus, each wireless network connection can be seen as a dependency to its parent station as we work our way

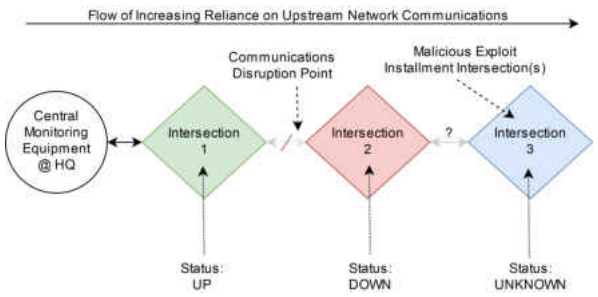


Figure 25: Diversionary cabinet access tactic. The circle on the left most represents the central headquarter. An attacker can disable the communication between intersections 1 and 2, and conduct the malicious exploitation at intersection 3 where is a few miles away.

further from the centrally headquartered location.

Consequently, a diversionary tactic would seriously affect the troubleshooting process of the traffic signal system. For example, one would covertly or explicitly break upstream network communications, thus leaving downstream traffic intersections with no network access to the rest of the traffic network. This would disable any sort of central monitoring including surveillance cameras and cabinet door alarms as there would be no network path to these devices. Figure 25 shows the diversionary cabinet access tactic.

To achieve the needed disruption to the network, one of the methods is to use radio frequency jamming techniques since the wireless 802.11 equipment is widely used to connect vehicle traffic networks [62]. As shown in works by Grover et al. [190] and Pelechrins et al. [101], 802.11 networks can be completely or selectively jammed to block communications between end devices. The use case for us would be to disrupt the communications pathway between a selected vehicle intersection and the traffic control master server located miles away. In reality, our partnering municipality informs us that network outages are already a common occurrence due to the interference generated by the deployment of wireless 802.11 access points in homes and business that are near traffic intersections. In

short, they would likely disregard our radio frequency disruption and jamming attacks as another common case of co-channel interference.

To traffic system monitoring staff, the statuses of intersections that lay on the other side of the network disruption or breakage would fundamentally be unknown due to the lack of network communications to the downed intersections. It is at this point that an attacker would access one of the downstream traffic cabinets with an unknown status. Throughout the period of unknown status, the attacker would have completely unmonitored access to the cabinet.

At some point, the municipality will have to troubleshoot the outage. We learn through our partnering municipality that the troubleshooting process could occur anywhere between instantaneously and 64 hours (if the attack is orchestrated outside of normal business hours during the weekend). Upon inspection, the maintenance staff would focus on the direct location of the network outage itself and not any of the unknown status intersections behind the disrupted connection. Once they managed to resolve the disruption at the first disrupted intersection, it is unlikely that they would investigate any of the previously unknown status intersections if all network communications return to normal.

Implication 7: The troubleshooting process of the real-world traffic signal systems makes it possible for the attacker to achieve stealthy access/control to the system..

7.4 Attacks Implementation and Testing

To learn the impact of the implications discussed in Section 7.3, we have crafted several attacking scenarios in which we test with our partnering municipality.

Figure 26: Traffic signal system in the municipality test lab. The left side shows a group of Model 2070 ATCs. The right top of the figure shows the traffic signal bulbs while the right bottom of the figure shows the CMU-212.

7.4.1 Environment Setup

We first partner with a local municipality to gain access to their traffic signals test lab, which is equipped with ITS cabinets, Intelight Model 2070 ATCs [121], and CMU-212 [81]. This lab is a mock-up of their operational traffic network and is used for their own testing and burn-in of equipment before deploying the devices to the field. The devices that were used in the lab are shown in Figure 26. The Intelight Model 2070 ATC is running the Linux 2.6.39 kernel as specified by ATC standard [2].

Moreover, we obtain a TS-2 cabinet and set up an environment that fulfills the NEMA standard. In this cabinet, the widely used Siemens Model 60 ATC [230] and EDI MMU-16LE [80] are deployed. The entire traffic signal system is shown in Figure 27 Like the Intelight Model 2070 ATC, the Siemens Model 60 also runs upon a Linux 2.6.39 kernel as specified by the ATC Standard.

7.4.2 Thread Model

We assume the target traffic signal system follows the ITS cabinet standard or the TS-2 cabinet standard. In both standards, we assume the ATC deployed inside the cabinet

Figure 27: Traffic Signal System of TS-2 Standard. The ① is a vehicle detection and surveillance system. The ② shows the MMU-16LE while the ③ shows the Siemens Model 60 ATC. The ④ and ⑤ indicate the Load Switch relays and traffic signal bulbs, respectively.

follows the ATC standard released by AASHTO, ITE, and NEMA. In our attack, we assume the access to the traffic signal system is gained via *Implications 1-4*. Specifically, in most scenarios, we only require the remote access achieved by *Implication 4*. In the all-direction green light attack, we provide two different attacking policies with physical access gained by *Implications 1-3* and remote access gained by *Implication 4*, respectively.

7.4.3 Attack Scenarios

Stealthy Manipulation and Control

As demonstrated in previous research [62, 144, 63], the monitoring and control of the traffic signal system could be used in a series of attacks such as Denial of Service (DoS) and causing traffic congestion. However, previous attack approaches control the traffic signals by either changing the configuration of the ATC or by injecting messages to the transportation system that are easy to be detected. For example, the transportation

engineers may simply pull the configuration of the ATC remotely to identify the abnormal configuration.

In our attack, we achieve a stealthy manipulation and control via intercepting the communication between the ATC and Load Switch relays that control the traffic signal lights. As discussed in Section 7.3.2 and *Implication 5*, the in-cabinet device communication between the ATC and traffic lights is performed via the serial port `/dev/sp3s` and `/dev/sp5s`, and the communication requires no authentication mechanism. To monitor and manipulate the communication, we replace the driver of these two devices in the system with a customized driver, and the customized driver records/modifies the message sent to the serial port before it is transmitted to the hardware.

With the customized driver, our attack is launched with a stealthy style since it modifies no configuration of the ATC and involves no additional messages. For example, we can increase the duration of the red light to introduce a traffic congestion. More serious congestion would be caused if we place all the traffic signals into a flashing red style. Even worse, malicious signal patterns such as all-direction flashing yellow may spark a critical accident. According to *Implication 7*, existing troubleshooting process of the traffic signal system can hardly detect the attack if the malicious traffic signal pattern is carefully designed.

Ransomware Deployment

One of the most crippling scenarios for a traffic network is the deployment of ransomware across all traffic control devices contained in the network. Using methodologies as described in Section 7.3, we design the most simplistic path towards a ransomware de-

ployment across a traffic network. In this scenario, all ATCs on a traffic control network would have their internal traffic control program processes disabled and all root login access to the internal Linux operating system would be denied, thus, each ATC would be held at ransom.

As specified by the ATC specification [2], the ATC stores all startup instructions in the Linux `/etc/inittab` daemon file. While investigating this file, we found an instruction to launch a shell script file that handles setting up the runtime environment and processes for the traffic control software. If one is to remove this shell script file, it completely disables the traffic control software from launching thus leaving its respective intersection uncontrolled. Rebooting the ATC devices will not resolve the issue, and the only way to resume the traffic control software is to replace the correct script that is responsible for launching the traffic control software. To further consolidate the attack, we can change the credentials of the SSH connection to prevent the transportation engineers from accessing the ATC system.

To extend the attack, we launch a ransomware deployment Python script in the partnered lab, which includes a large number of Intelight 2070 ATCs on the test traffic network. With the script, we are able to make a list containing IPs of all known ATCs on the traffic network then deploy our ransomware engagement shell commands issued over SSH.

The Destruction. In the network of the road agency that we partnered with, this exploit would allow us to take control of 400 ATCs running the known traffic control software. To fix a ransomware affected traffic signal system at an intersection, a transportation engineer would need to drive to the intersection and physically update the firmware of the ATC. If we assume that it will take 1 hour to fix each ATC (it might take more time because of

the traffic congestion and none of the ATC traffic signals operating), and assuming that 10 workers have the expertise to do this. The time for resuming the complete traffic signal system would be: 400 controllers * 1 hour / 10 engineers = 40 hours/engineer = 1 week (if an engineer works 8 hours/day). If we assume that an engineer is paid \$$40$/hour, the estimated cost for fixing this would be 400 controllers * 1 hour * \$$40$/hour = \$$16,000$ USD. A previous study [51] also shows that simply reconfiguring the timings of 60 intersections in one district of Boston could save \$$1.2$ million per year. Additionally, we also identify that many states (e.g., California, Florida, Michigan, Missouri, Ohio, Oregon, South Carolina, Texas, Virginia, Wisconsin, etc.) currently use the 2070 ATC [250, 47, 235], which means that our attacks might be deployed in these states as well.

All-Direction Green Lights

When considering the most dangerous state for an intersection, we conceived the idea of all-direction green lights. An intersection displaying green lights in all directions would leave drivers defenseless to vehicle cross traffic traveling at speed as they passed through. In order to make this happen, one would have to override the fail-protection of the MMU/CMU, then program the all-direction green light pattern into the traffic controller. We chose to investigate this possibility heavily as the MMU/CMU was not shown to be tested in previous work.

The MMU/CMU plays the role of policing traffic patterns shown by the ATC. If a traffic pattern is displayed that would be dangerous, such as all-direction greens, the MMU/CMU steps in and places the intersection into conflict flash. Furthermore, if serial communication fails amongst any of the traffic cabinet's devices, the intersection is placed into conflict

Figure 28: CMU-212 display unit showing the Datakey configuration for USER ID and MONITOR ID which was written using the home-made Arduino Datakey writer to allow for full-permissive configuration.

flash. This made for a difficult process as any event that placed the cabinet ecosystem out-of-balance would trigger a conflict flash state. Due to the differences in attack policies, we discuss this attack with and without physical access, respectively.

With Physical Access. As discussed in Section 7.3.3, the configuration data such as unsafe states is defined by the Programming Card and Datakey in MMU and CMU, respectively. To overcome accidentally triggering conflict states, we can directly override the configuration of the MMU/CMU with physical access according to *Implication 6*. Since the configuration of the Programming Card is simply achieved by soldered wire jumpers, here we only show how to override the configuration in the CMU Datakey.

While we find the CMU's specification for the address layout and configuration parameters for the Datakey in the ITS Cabinet Specification [115], we believe that the parameter selection would be difficult for someone without traffic device configuration experience. In order to combat this, we look for configuration generation programs on the CMU manufacturers website. It does not take us long to find one as we quickly discovered a free program [82] offered which would allow us to create the configuration files for the Datakey using a wizard-style approach. This wizard would handle parameter setup, leaving us to only to configure the nullification of conflict states for the intersection which was as simple as selecting a group of checkboxes called `permissives`. A `permissive` is a setting

that which specifies what individual traffic signal light bulb is permitted to be turned on with each other light bulb of the intersection. This is done as a method to prevent two cross-directions of travel from receiving concurrent green lights which would cause passing vehicles to enter a potentially dangerous situation. If two signal light bulbs try to turn on that is not set to be permissive with each other, the CMU will engage and place the intersection into conflict flash. After using the key generation program to generate a configuration file allowing for all-direction green permissives, we use our Arduino Uno LCK4000 flasher to write the configuration to the key as shown in Figure 24. Figure 28 shows the CMU Datakey configuration on a display screen. This Datakey is written via the home-made Arduino writer using our generated key file.

The last step in configuring all-direction greens lights is to place the correct settings in the ATC traffic control program. However, during our experiment, it is discovered that the Intelight Model 2070 ATC must maintain nearly constant contact with the CMU over serial communications, and this contact periodically shares the configuration of the LCK4000 Datakey and the ATC with each other. If the configurations do not match, the CMU will trigger a conflict flash. To combat this issue, the traffic controller must be configured to match the all-direction green permissive configuration on the CMU.

In order to set up the ATC with a matching configuration to the CMU, all that we required is the front panel controls and display screen located directly on the unit. Navigating through the front panel menu controls, we find that the traffic control software features a similar parameter setup to what we saw on the CMU. In this menu, we are able to explicitly state the permissives of the intersection then construct an all-directions green traffic pattern. We are then able to schedule for an all-directions green pattern to run

Figure 29: All-direction green lights being displayed on traffic signal test equipment. The left 4 green LEDs represent the through directions of travel at an intersection while the right 4 green LEDs represent the corresponding left-turn lanes.

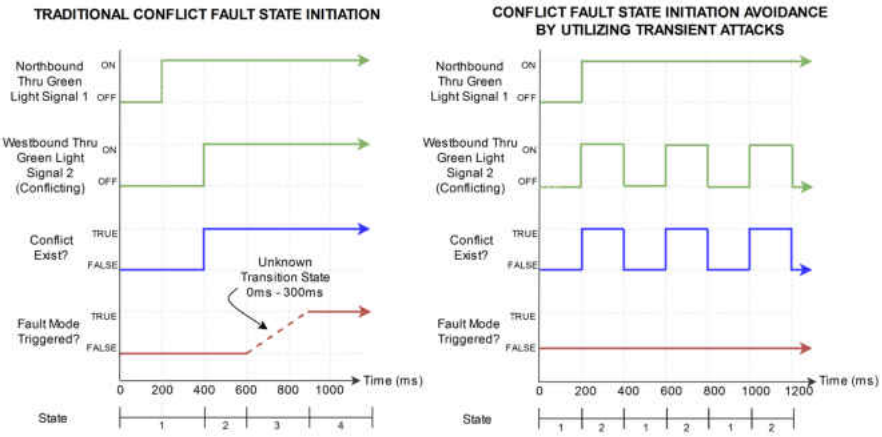


Figure 30: Comparison of a typical all-directions green conflict flash state initiation versus transient avoidance attack tactic.

in another menu. Shortly after scheduling the pattern to run and waiting for the transition to occur, we are greeted with the all-directions green configuration. A test displaying all-direction greens is shown in Figure 29.

With Remote Access. Although the configuration of the ATC could be modified via remote access, the aforementioned approach requires physical access to reconfigure the unsafe states of MMU/CMU. Since the configuration in devices like the MMU programming card is achieved by soldered wire jumpers, it would be difficult to override the configuration without physical access.

To bypass the fail-safe units, we implement an attack called the transient avoidance attack tactic. The root feature of this attack is that the fail-safe unit does not trigger a conflict state until conflicting control signals exist for 200 milliseconds or greater. Following this 200 millisecond wait period, the fail-safe requires up to an additional 300 milliseconds to place the intersection into a conflict state (all-directions flashing red lights). Figure 30 shows the details of the transient attack. From the top, the first line and second lines represent the on/off signal of two green lights that conflict at an intersection. The third line represents the presence on a conflict situation. The fourth line displays if an intersection has entered a conflict flash failure state. The state designation, seen on the bottom of the graphs, is described as the following: 1) An intersection is in a conflict free running state; 2) A conflict has occurred. The conflicting signals must exist for 200 milliseconds before triggering a conflict state; 3) Conflicting signals have been shown for more than 200 milliseconds. In the next 300 milliseconds timespan, the fail-safe unit must place the intersection into a conflict flash state; 4) The intersection is currently in the conflict flash state.

Another challenge that we will have overcome is the fail-safes' use of Recurrent Pulse Detection (RPD) [81, 80]. This mechanism is used to detect failures resulting in voltage leaks from a traffic signal's Load Switch relays. This mechanism looks for voltage leaks lasting 1 to 200 milliseconds and triggers a conflict flash state if they meet a certain criteria level in regards to power, duration, and frequency. In practice, our experiment shows that the RPD mechanism will not trigger a fault if an off time of 24 milliseconds or greater duration is used to separate conflicting signals such as in the procedure of triggering each green light bulb during the all-directions green attack. Note that the transient attack places

the traffic lights into a flicker status. Considering the high flicker frequency, the influence of the real-world ambient light, and the long distance between the real-world traffic lights and the drivers, the flickering green lights are likely to be recognized as constant green lights.

CHAPTER 8 CONCLUSION AND FUTURE WORK

8.1 Conclusion

In recent years, ARM platform becomes common in different areas including mobile phones, tablets, Internet of Things (IoT) devices, and even cloud platforms. However, the research on the security of ARM platform is far from enough comparing to that on the security of the traditional x86 platform. In my dissertation, I designed and implemented software-based and hardware-based approaches to secure the ARM platform.

First, I design and implement DexLego [174], a program transformation system that reveals the hidden code in Android applications and transfers them to analyzable pattern via instruction-level extracting and reassembling. DexLego collects bytecode and data when they are executed and accessed, and reassembles the collected result into a valid DEX file for static analysis tools. Since DexLego extracts all executed instructions, it is able to uncover the malicious behaviors of the packed applications or malware with self-modifying code. In DexLego, I design a novel reassembling approach to reconstruct the entire executed control flows including self-modifying code and implement the first prototype of force execution on Android to improve the code coverage. I evaluate DexLego on real-world packed applications and DroidBench samples. The evaluation results show that DexLego successfully unpacks and reconstructs the behavior of the applications. The F-measures (i.e., analysis accuracy) of state-of-the-art static analysis tools have increased for more than 20% with the help of DexLego. The code coverage experiments show that the force execution module helps to increases the coverage of state-of-the-art fuzzing tools from 32% to 82%.

Second, I design Ninja [173, 175], a transparent malware analysis framework on ARM platform based on hardware features including TrustZone technology, Performance Monitoring Unit (PMU), and Embedded Trace Macrocell (ETM). I implement a prototype of Ninja that embodies a trace subsystem with different tracing granularities and a debug subsystem with a GDB-like debugging protocol on ARM Juno development board. Additionally, hardware-based traps and memory protection are leveraged to keep the use of system registers transparent to the target application. The experiment results show that Ninja can transparently monitor and analyze the behavior of the malware samples. Moreover, Ninja introduces reasonable overhead. I evaluate the performance of the trace subsystem with several popular benchmarks, and the result shows that the overheads of the instruction trace and system call trace are less than 1% and the Android API trace introduces 4 to 154 times slowdown.

Third, based on the security analysis of the ARM debugging features, I design a novel attack scenario, which I call Nailgun [176]. Nailgun works on a processor running in a low-privilege mode and accesses the high-privilege content of the system without restriction via the inter-processor debugging model. Specifically, with Nailgun, the low-privilege processor can trace the high-privilege execution and even execute arbitrary payload at a high-privilege mode. I implement Nailgun on commercial devices with different SoCs and architectures, and the experiment results show that Nailgun is able to break the privilege isolation enforced by the ARM architecture. The experiment also shows that Nailgun can leak the fingerprint image stored in TrustZone from the commercial mobile phone.

To learn more about the security aspect of existing TEEs, I also summarize the technique behind the widely deployed TEEs and propose the challenges that these TEEs would have

to face [178]. Additionally, the potential adoption of existing TEEs to the emerging edge computing is present [172]. Additionally I perform a study on the security of deployed traffic signal infrastructure [177] and show that remotely controlling a traffic signal and bypassing the fail-safe mechanism is possible.

8.2 Future Work

ARM platforms are complex systems, and the security of them is far from a solved problem. The research I have done only represents a few aspects of this field and much more research is still required. In my future research, I will pursue three areas.

First, I will continue working on transparent malware analysis. Although the prototype of Ninja has improved the transparency of malware analysis systems, it is still not perfect. For instance, it still shares the main CPU with the target program, which might leave detectable traces that reveal the presence of the analysis system via cache side-channels. Furthermore, the usage of hardware components relies on I/O registers, and these registers may be accessed by the malware from the main CPU. In the x86 architecture, co-processors are widely deployed to aid the management and firmware updating. For example, the Integrated Dell Remote Access Controller (iDRAC) developed by Dell is based on the Intel Management Engine (ME), which is a micro-computer embedded inside of all recent Intel processors. In the ARM architecture, the co-processor also widely exists. Since it is a separated processor that does not share cache or time slice with the main processor, the instruction execution on this processor is transparent to the programs running in the main processor. Thus, I plan to leverage the co-processors in the SoCs to achieve the malware analysis and protect the I/O registers via system bus monitoring and manipulation.

Second, I will investigate the security aspects of other hardware components in the SoC systems. Hardware debugging features were trustworthy in the traditional debugging model, but it turns to be vulnerable when the advanced multi-core systems and inter-processor debugging are involved. It offers me an insight that the deployment of new and advanced systems may impact the security of a legacy mechanism. In the modern computer system, there exists a large number of hardware components/features, but the security of these components/features is not completely clear. For example, recent Meltdown and Spectre attacks are caused by the speculative execution feature of the processor, which has been developed and trusted for a long time. Thus, I plan to perform security analysis to more well-known hardware components (e.g., DMA and MMU) as well as newly announce hardware features (e.g., Pointer Authentication Code in ARM v8.3 and Memory Tag in ARM v8.5). Additionally, I will also work on an effective defense mechanism to Nailgun attack.

Third, the emerging of Machine Learning and Artificial Intelligence raises my interest in the GPU. On one hand, several attacks on GPU have been proposed by researchers, and how to guarantee the security of the GPU is still an open problem. Existing GPU computing interface (e.g., Nvidia CUDA and ARM Compute Library) offers the developers task-based SDKs, but the isolation between tasks still need to be carefully examined. On the other hand, the GPU is naturally isolated from the main processor, and applications running in the GPUs would not have to trust the main processor and any software running on it, which is essential for a TEE. Additionally, similar to the co-processors, the separated memory and cache makes GPU to be immune to a range of side-channel attacks. Thus, I believe there is a potential to make GPU serve as a Trusted Execution Environment.

REFERENCES

[1] AASHTO. American association of state highway and transportation officials. https://www.transportation.org/, 2017.

[2] AASHTO, ITE, and NEMA. Advanced Transportation Controller (ATC) standard version 06. https://www.ite.org/pub/?id=acaf6aca-d1fd-f0ec-86ca-79ad05a7cab6, 2018.

[3] A. Abbasi, T. Holz, E. Zambon, and S. Etalle. ECFI: Asynchronous control flow integrity for programmable logic controllers. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC'17)*, 2017.

[4] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. C-FLAT: Control-flow attestation for embedded systems software. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, 2016.

[5] Alibaba Inc. AliProtector. http://jaq.alibaba.com/.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, 2012.

[7] AMD. BIOS and kernel developer's guide for AMD family 16h models 30h-3Fh processors. http://support.amd.com/TechDocs/52740_16h_Models_30h-3Fh_BKDG.pdf.

[8] AMD. Architecture programmer's manual volume 2: System programming. https:

//support.amd.com/TechDocs/24593.pdf, 2017.

[9] AMD. AMD EPYC 7251 processor. https://www.amd.com/en/products/cpu/amd-epyc-7251, 2018.

[10] AMD TATS BIOS Development Group. AMD security and server innovation. http://www.uefi.org/sites/default/files/resources/UEFI_PlugFest_AMD_Security_and_Server_innovation_AMD_March_2013.pdf.

[11] AmishTech. Motorola e4 plus - More than just a fingerprint reader. https://community.sprint.com/t5/Android-Influence/Motorola-E4-Plus-More-Than-Just-a-Fingerprint-Reader/ba-p/979521.

[12] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'12)*, 2012.

[13] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*, 2013.

[14] Android APK Encryption and Protection. APKProtector. https://sourceforge.net/projects/apkprotect/, 2013.

[15] Anubis. Analyzing unknown binaries. http://anubis.iseclab.org.

[16] Apple. HomeKit. https://developer.apple.com/homekit/.

[17] Arduino. Starter kit. https://store.arduino.cc/usa/arduino-starter-kit, 2017.

[18] ARM. Architecture reference manual ARMv7-A and ARMv7-R edition. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406

c/index.html.

[19] ARM. ARMv6-M architecture reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0419e.

[20] ARM. ARMv8-A reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.k/index.html.

[21] ARM. CoreLink NIC-400 network interconnect technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0475g/index.html.

[22] ARM. CoreSight components technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0314h/index.html.

[23] ARM. CoreSight trace memory controller technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0461b/DDI0461B_tmc_r0p1_trm.pdf.

[24] ARM. DS-5 development studio. https://developer.arm.com/products/software-development-tools/ds-5-development-studio.

[25] ARM. Dual-Timer module (SP804) technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0271d/DDI0271.pdf.

[26] ARM. Embedded trace macrocell architecture specification. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/index.html.

[27] ARM. Generic interrupt controller architecture specification. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0048b/index.html.

[28] ARM. Juno ARM development platform SoC technical reference manual. https://www.arm.com/files/pdf/DDI0515D1a_juno_arm_development_pla

tform_soc_trm.pdf.

[29] ARM. PrimeCell real time clock technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0224b/DDI0224.pdf.

[30] ARM. Trusted firmware. https://github.com/ARM-software/arm-trusted-firmware.

[31] ARM. TrustZone security. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html.

[32] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux containers with Intel SGX. In *Proceedings of The 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[33] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, 2014.

[34] ATEK Access Technologies. Datakey microwire protocol specification. http://datakey.com/downloads/223-0017-003_REVI_MWInterfaceSpec_SBM.pdf, 2014.

[35] ATEK Access Technologies. Datakey LCK series specification sheet. http://datakey.com/downloads/LCK_Series_DS_REV.D.pdf, 2015.

[36] ATEK Access Technologies. Access the power of technology. http://atekcompanies.com/access-technologies, 2017.

[37] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin. ANVIL: Software-based protection against next-generation rowhammer attacks. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, 2016.

[38] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, 2014.

[39] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. HyperSentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[40] A. M. Azab, P. Ning, and X. Zhang. SICE: A hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, 2011.

[41] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *Proceedings of the 19th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'13)*, 2013.

[42] Baidu Inc. BaiduProtector. `http://app.baidu.com/jiagu/`.

[43] D. Balzarotti, G. Banks, M. Cova, V. Felmetsger, R. Kemmerer, W. Robertson, F. Valeur, and G. Vigna. An experience in testing the security of real-world electronic voting systems. *IEEE Transactions on Software Engineering*, 2010.

[44] Bangcle Ltd. BangcleProtector. https://www.bangcle.com/, 2013.

[45] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *Proceedings of the 30th Annual International Conference on Automated Software Engineering (ASE'15)*, 2015.

[46] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[47] Bid Contract. Search federal, state, and local government contracts, government bids, and RFPs. https://www.bidcontract.com/government-contracts-bids/search-government-Bids-Contracts.aspx?s=2070&t=FE&is=0, 2017.

[48] Bluebox Security Inc. Android security analysis challenge: Tampering Dalvik bytecode during runtime. https://bluebox.com/android-security-analysis-challenge-tampering-dalvik-bytecode-during-runtime/.

[49] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[50] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry. CoDisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[51] Boston Transportation Department. The benefits of retiming/rephasing traffic signals in the back bay. https://www.cityofboston.gov/images_documents/The%2

0Benefits%20of%20Traffic%20Signal%20Retiming%20Report_tcm3-18554.pdf, 2010.

[52] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A.-R. Sadeghi. Regulating ARM TrustZone devices in restricted spaces. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'16)*, 2016.

[53] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT'17)*, 2017.

[54] BSDaemon, coideloko, and D0nAnd0n. System management mode hack: Using SMM for other purposes. *Phrack Magazine*, 2008.

[55] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for Android. In *Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)*, 2011.

[56] J. Butterworth, C. Kallenberg, and X. Kovah. BIOS Chronomancy: Fixing the core root of trust for measurement. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*, 2013.

[57] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[58] S. Calzavara, I. Grishchenko, and M. Maffei. HornDroid: Practical and sound static analysis of Android applications by SMT solving. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P'16)*, 2016.

[59] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework. In *Proceedings of 22nd Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[60] C. Cerrudo. Hacking US traffic control systems. In *DEFCON*, 2014.

[61] Chainfire. CF-Bench. `https://play.google.com/store/apps/details?id=eu.chainfire.cfbench`.

[62] Q. A. Chen, Y. Yin, Y. Feng, Z. M. Mao, and H. X. Liu. Green lights forever: Analyzing the security of traffic infrastructure. In *Proceedings of the 8th USENIX Workshop on Offensive Technologies (WOOT'14)*, 2014.

[63] Q. A. Chen, Y. Yin, Y. Feng, Z. M. Mao, and H. X. Liu. Exposing congestion attack on emerging connected vehicle based traffic signal control. In *Proceedings of 25th Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[64] Y. Chen, Q. Feng, and W. Shi. An industrial robot system based on edge computing: An early experience. In *Proceedings of USENIX Workshop on Hot Topics in Edge Computing (HotEdge'18)*, 2018.

[65] S. Clark, T. Goodspeed, P. Metzger, Z. Wasserman, K. Xu, and M. Blaze. Why (special agent) johnny (still) can't encrypt: A security analysis of the APCO project 25 two-way radio system. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security'11)*, 2011.

[66] L. Cojocar, K. Razavi, and H. Bos. Off-the-shelf embedded devices as platforms for security research. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*, 2017.

[67] B. Confais, A. Lebre, and B. Parrein. Performance analysis of object store systems in a fog and edge computing infrastructure. *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXIII*, 2017.

[68] Coreboot. Open-Source BIOS. `http://www.coreboot.org/`.

[69] N. Corteggiani, G. Camurati, and A. Francillon. Inception: System-wide security testing of real-world embedded systems software. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*, 2018.

[70] V. Costan and S. Devadas. Intel SGX Explained. `https://eprint.iacr.org/2016/086.pdf`.

[71] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti. Understanding Linux malware. In *Proceedings of 39th IEEE Symposium on Security and Privacy (S&P'18)*, 2018.

[72] C. Dall and J. Nieh. KVM/ARM: The design and implementation of the linux ARM hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, 2014.

[73] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro. DroidScribe: Classifying Android malware based on runtime behavior. *Mobile Security Technologies (MoST'16)*, 2016.

[74] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th ACM/IEEE International Symposium on Computer Architecture (ISCA'13)*, 2013.

[75] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu. iRiS: Vetting private api abuse in iOS applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[76] Z. Deng, X. Zhang, and D. Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)*, 2013.

[77] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, 2008.

[78] L. Duflot, O. Levillain, B. Morin, and O. Grumelard. System management mode design and security issues. `http://www.ssi.gouv.fr/IMG/pdf/IT_Defense_2010_final.pdf`.

[79] L. Duflot, O. Levillain, B. Morin, and O. Grumelard. Getting into the SMRAM: SMM reloaded. In *Proceedings of the 12th CanSecWest Conference (CanSecWest'09)*, 2009.

[80] Eberle Design, Inc. MMU-16LE series SmartMonitor. `https://www.editraffic.com/wp-content/uploads/888-0116-001-MMU-16LE-Operation-Manual.pdf`, 2012.

[81] Eberle Design, Inc. CMU-212. `https://www.editraffic.com/wp-content/uploads/888-0212-001-CMU-212-Operation-Manual.pdf`, 2015.

[82] Eberle Design, Inc. Traffic control software. `https://www.editraffic.com/support-traffic-control-software/`, 2016.

[83] EC SPRIDE Secure Software Engineering Group. DroidBench. `https://github.com/secure-software-engineering/DroidBench`.

[84] S. Embleton, S. Sparks, and C. Zou. SMM rootkits: A new breed of OS independent malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks (SecureComm'08)*, 2008.

[85] Enck, William and Gilbert, Peter and Cox, Landon P and Jung, Jaeyeon and McDaniel, Patrick and Sheth, Anmol N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDIâĂŹ10)*, 2010.

[86] F-Droid. F-Droid. `https://f-droid.org/`, 2011.

[87] F-Droid. F-Droid Random Page. `https://f-droid.org/wiki/page/Special:Random`, 2011.

[88] Fingerprints. FPC1020 touch sensor. `https://www.fingerprints.com/technology/hardware/sensors/fpc1020/`.

[89] Fingerprints. Product specification FPC1020. `http://www.shenzhen2u.com/doc/Module/Fingerprint/710-FPC1020_PB3_Product-Specification.pdf`.

[90] Fourth Dimension Traffic. The D4 traffic signal controller software. `https://fourthdimensiontraffic.com/about/about.html`, 2017.

[91] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. A. Mohammed, and S. A. Zonouz. Hey, my malware knows physics! Attacking PLCs with physical model aware rootkit. In *Proceedings of 24th Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[92] D. Goodin. The hijacking flaw that lurked in Intel chips is worse than anyone thought. `https://arstechnica.com/security/2017/05/the-hijacking-flaw-`

that-lurked-in-intel-chips-is-worse-than-anyone-thought/.

[93] Google. Google play. https://play.google.com/store?hl=en, 2017.

[94] Google Inc. Android open source project. https://source.android.com/.

[95] Google Inc. Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.

[96] Google Inc. Dalvik executable format. https://source.android.com/devices/tech/dalvik/dex-format.html.

[97] Google Inc. MonkeyRunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[98] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of Android applications in DroidSafe. In *Proceedings of 22nd Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[99] G. Grassi, K. Jamieson, P. Bahl, and G. Pau. Parkmaster: An in-vehicle, edge-based video analytics service for detecting open parking spaces in urban environments. In *Proceedings of the 2nd ACM/IEEE Symposium on Edge Computing (SEC'17)*, 2017.

[100] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth. AutoLock: Why cache attacks on ARM are harder than you think. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*, 2017.

[101] K. Grover, A. Lim, and Q. Yang. Jamming and anti–jamming techniques in wireless networks: a survey. *International Journal of Ad Hoc and Ubiquitous Computing*, 2014.

[102] A. Grush. Huawei unveils big ambitions with the 6-inch Huawei ascend mate 7. https://consumer.huawei.com/nl/press/news/2014/hw-413119/.

[103] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. TrustShadow: Secure execution of unmodified applications with ARM trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*, 2017.

[104] Hackster. Raspberry PI IoT projects. `https://www.hackster.io/raspberry-pi/projects`.

[105] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile systems, applications, and services (MobiSys'14)*, 2014.

[106] Hisilicon. Kirin processors. `http://www.hisilicon.com/en/Products`.

[107] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, V. Phegade, and J. del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*, 2013.

[108] Z. Hua, J. Gu, Y. Xia, and H. Chen. vTZ: Virtualizing ARM TrustZone. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*, 2017.

[109] Huawei. Ascend mate 7. `https://consumer.huawei.com/en/support/phones/mate7/`.

[110] Huawei. Open source release center. `https://consumer.huawei.com/en/opensource/`.

[111] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of 11th USENIX Symposium*

on Operating Systems Design and Implementation (OSDI'16), 2016.

[112] C.-C. Hwang. ptm2human. https://github.com/hwangcc23/ptm2human.

[113] J. hyuk Jung and J. Lee. DABID: The powerful interactive Android debugger for Android malware analysis.

[114] Ijiami Inc. IJiamiProtector. http://www.ijiami.cn/AppProtect.

[115] Institute of Transportation Engineers. Standard specification for roadside cabinets. https://www.ite.org/pub/E26A4960-2354-D714-51E1-FCD483B751AA, 2006.

[116] Intel. 64 and IA-32 architectures software developer manuals. https://software.intel.com/en-us/articles/intel-sdm.

[117] Intel. ISCA 2015 SGX tutorial. https://software.intel.com/sites/default/files/332680-002.pdf.

[118] Intel. Fog reference design overview. https://www.intel.com/content/www/us/en/internet-of-things/fog-reference-design-overview.html, 2017.

[119] Intel. SGX SDK. https://software.intel.com/en-us/sgx-sdk/, 2017.

[120] Intel Security Group. INTEL-SA-00075. https://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00075&languageid=en-fr.

[121] Intelight. 2070 ATC controllers. https://www.intelight-its.com/product-categories/2070-type-controllers/, 2017.

[122] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *Proceedings of 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[123] G. Irazoqui, T. Eisenbarth, and B. Sunar. Cross processor cache attacks. In *Proceedings of the 11th ACM SIGSAC Symposium on Information, Computer and Communications Security (AsiaCCS'16)*, 2016.

[124] ITE. About ITE. `https://www.ite.org/about-ite/about-ite/`, 2017.

[125] JaCoCo. Java Code Coverage Library. `http://www.eclemma.org/jacoco/`, 2009.

[126] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion. Sok: Introspections on trust and the semantic gap. In *Proceedings of 35th IEEE Symposium on Security and Privacy (S&P'14)*, 2014.

[127] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han. OpenSGX: An open platform for SGX research. In *Proceedings of 23rd Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[128] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang. PrivateZone: Providing a private execution environment using ARM TrustZone. *IEEE Transactions on Dependable and Secure Computing*, 2016.

[129] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. In *Proceedings of 22nd Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[130] S. Jin, J. Seol, J. Huh, and S. Maeng. Hardware-assisted secure resource accounting under a vulnerable hypervisor. In *Proceedings of 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environment (VEE'15)*, 2015.

[131] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: automatically generating heuristics to detect Android emulators. In *Proceedings of the 30th Annual Computer Secu-*

rity Applications Conference (ACSAC'14), 2014.

[132] J. Jongma. SuperSU. https://chainfire.eu/.

[133] C. Kallenberg and X. Kovah. How many million BIOSes would you like to Iinfect? http://conference.hitb.org/hitbsecconf2015ams/wp-content/uploads/2015/02/D1T1-Xeno-Kovah-and-Corey-Kallenberg-How-Many-Million-BIOSes-Would-You-Like-to-Infect.pdf.

[134] C. Kallenberg and R. Wojtczuk. Speed racer: Exploiting an Intel flash protection race condition. https://events.ccc.de/congress/2014/Fahrplan/system/attachments/2565/original/speed_racer_whitepaper.pdf.

[135] D. Kaplan. AMD x86 memory encryption technologies. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kaplan.

[136] D. Kaplan, J. Powell, and T. Woller. AMD memory encryption. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[137] D. Kaplan, T. Woller, and J. Powell. AMD memory encryption tutorial. https://sites.google.com/site/metisca2016/.

[138] V. Karande, E. Buaman, Z. Lin, and L. Khan. SGX-log : Securing system logs with SGX. In *Proceedings of the 12th ACM on Asia Conference on Computer and Communications Security (AsiaCCS'17)*, 2017.

[139] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu. J-Force: Forced execution on JavaScript. In *Proceedings of the 26th International Conference on World Wide Web (WWW'17)*, 2017.

[140] Kirat, Dhilung and Vigna, Giovanni. MalGene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[141] Kirat, Dhilung and Vigna, Giovanni and Kruegel, Christopher. Barecloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*, 2014.

[142] K. Koscher, T. Kohno, and D. Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *Proceedings of the 9th USENIX Workshop on Offensive Technologies (WOOT'15)*, 2015.

[143] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: A retrospective. In *Proceedings of the Cetus Users and Compiler Infastructure Workshop (CETUS'11)*, 2011.

[144] A. Laszka, B. Potteiger, Y. Vorobeychik, S. Amin, and X. Koutsoukos. Vulnerability of transportation networks to traffic-signal tampering. In *Proceedings of the 7th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS'16)*, 2016.

[145] K. Leach, C. Spensky, W. Weimer, and F. Zhang. Towards transparent introspection. In *Proceedings of 23rd IEEE Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, 2016.

[146] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek. Towards a practical solution to detect code reuse attacks on ARM mobile devices. In *Proceedings of the 4th Workshop on Hardware and Architectural Support for Security and Privacy (HASP'15)*, 2015.

[147] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Pro-*

ceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14), 2014.

[148] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1 (ICSE'15)*, 2015.

[149] W. Li, H. Li, H. Chen, and Y. Xia. AdAttester: Secure online mobile advertisement attestation using TrustZone. In *Proceedings of The 13th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*, 2015.

[150] Z. Li, D. Jin, C. Hannon, M. Shahidehpour, and J. Wang. Assessing and mitigating cybersecurity tisks of traffic light systems in smart cities. *IET Cyber-Physical Systems: Theory & Applications*, 2016.

[151] Licel Inc. DexProtector. `https://dexprotector.com/`.

[152] Linaro. ARM development platform software. `https://releases.linaro.org/members/arm/platforms/15.09/`.

[153] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*, 2015.

[154] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache attacks on mobile devices. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security'16)*, 2016.

[155] G. Little. NSA's ANT division catalog of exploits for nearly every major software/hardware/firmware. `https://leaksource.wordpress.com/2013/12/30/ns`

as-ant-division-catalog-of-exploits-for-nearly-every-major-software-hardware-firmware/.

[156] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[157] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC'13/FSE'13)*, 2013.

[158] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'16)*, 2016.

[159] R. Marek. AMD x86 SMU firmware analysis - Do you care about Matroshka processors? https://events.ccc.de/congress/2014/Fahrplan/system/attachments/2503/original/ccc-final.pdf.

[160] S. Mazloom, M. Rezaeirad, A. Hunter, and D. McCoy. A security analysis of an in-vehicle infotainment and app platform. In *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT'16)*, 2016.

[161] McAfee Labs. Mobile threat report. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-dec-2018.pdf.

[162] F. Mckeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative instructions and software model for isolated execu-

tion. In *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*, 2013.

[163] MediaTek. This chip powers mobile. https://www.mediatek.com/products/smartphones.

[164] Microsoft. Azure sphere. https://www.microsoft.com/en-us/azure-sphere/.

[165] miniNodes. ARM servers. https://www.mininodes.com/.

[166] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing Android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 2012.

[167] Motorola. E4 plus. https://www.motorola.com/us/products/moto-e-plus-gen-4.

[168] Motorola. Nexus 6. https://support.motorola.com/products/cell-phones/android-series/nexus-6.

[169] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna. BareDroid: Large-scale analysis of Android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*, 2015.

[170] National Electrical Manufacturers Association. Standards publication TS 2-2003. https://www.nema.org/Standards/ComplimentaryDocuments/Contents%20and%20Scope%20TS%202-2003%20(R2008).pdf, 2003.

[171] NEMA. National electrical manufacturers association. https://www.nema.org/pages/default.aspx, 2017.

[172] Z. Ning, J. Liao, F. Zhang, and W. Shi. Preliminary study of trusted execution environments on heterogeneous edge platforms. In *Proceedings of the 1st ACM/IEEE Workshop on Security and Privacy in Edge Computing (EdgeSP'18), in conjunction with the 3rd ACM/IEEE Symposium on Edge Computing (SEC'18)*, 2018.

[173] Z. Ning and F. Zhang. Ninja: Towards transparent tracing and debugging on ARM. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*, 2017.

[174] Z. Ning and F. Zhang. Dexlego: Reassembleable bytecode extraction for aiding static analysis. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18)*, 2018.

[175] Z. Ning and F. Zhang. Hardware-assisted transparent tracing and debugging on arm. *IEEE Transactions on Information Forensics and Security (TIFS)*, 14(6):1595–1609, 2018.

[176] Z. Ning and F. Zhang. Understanding the security of arm debugging features. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[177] Z. Ning, F. Zhang, and S. Remias. Understanding the security of traffic signal infrastructure. In *Proceedings of the 16th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'19)*, 2019.

[178] Z. Ning, F. Zhang, W. Shi, and W. Shi. Position paper: Challenges towards securing hardware-assisted execution environments. In *Proceedings of the 6th Hardware and Architectural Support for Security and Privacy (HASP'17), in conjunction with the 44th International Symposium on Computer Architecture (ISCA'17)*, 2017.

[179] NQ Mobile. NetQinProtector. `https://shield.nq.com`, 2014.

[180] NSF. TWC: Small: System infrastructure for SMM-based runtime integrity measurement. https://nsf.gov/awardsearch/showAward?AWD_ID=1528185.

[181] NXP. i.MX53 multimedia applications processor reference manual. https://cache.freescale.com/files/32bit/doc/ref_manual/iMX53RM.pdf.

[182] NXP. i.MX53 quick start board. https://www.nxp.com/docs/en/user-guide/IMX53QKSRTRQSG.pdf.

[183] Oleksandr Bazhaniuk and John Loucaides and Lee Rosenbaum and Mark R. Tuttle and Vincent Zimmer. Symbolic execution for BIOS security. In *Proceedings of 9th USENIX Workshop on Offensive Technologies (WOOT'15)*, 2015.

[184] Oleksandr Bazhaniuk, Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alexander Matrosov, Mickey Shkatov. A new class of vulnerabilities in SMI handlers. http://www.c7zero.info/stuff/ANewClassOfVulnInSMIHandlers_csw2015.pdf.

[185] OpenFog. Consortium. https://www.openfogconsortium.org/, 2017.

[186] OpenOCD. Open on-chip debugger. http://openocd.org/.

[187] OpenSSL Software Foundation. OpenSSL cryptography and SSL/TLS toolkit. https://www.openssl.org/.

[188] Packet. Cloud service. https://www.packet.net/.

[189] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin. Dark Hazard: Learning-based, large-scale discovery of hidden sensitive operations in Android apps. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[190] K. Pelechrinis, M. Iliofotou, and S. V. Krishnamurthy. Denial of service attacks in wireless networks: The case of jammers. *IEEE Communications Surveys & Tutorials*,

2011.

[191] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-Force: Force-executing binary programs for security applications. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*, 2014.

[192] C. Perez. Tenable blog: Rediscovering the Intel AMT vulnerability. `https://www.tenable.com/blog/rediscovering-the-intel-amt-vulnerability`.

[193] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of Android malware. In *Proceedings of the 7th European Workshop on System Security (EurSec'14)*, 2014.

[194] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC'10)*, 2010.

[195] Primate Labs. GeekBench. `https://www.geekbench.com/`, 2016.

[196] Primate Labs. GeekBench Android. `https://play.google.com/store/apps/details?id=com.primatelabs.geekbench`, 2018.

[197] B. Qi, L. Kang, and S. Banerjee. A vehicle-based edge computing platform for transit and human mobility analytics. In *Proceedings of the 2nd ACM/IEEE Symposium on Edge Computing (SEC'17)*, 2017.

[198] Qihoo 360 Inc. 360Protector. `http://jiagu.360.cn/protection`.

[199] Qihoo 360 Inc. 360 market. `http://zhushou.360.cn/`, 2017.

[200] Qualcomm. Snapdragon processors. `https://www.qualcomm.com/products/mobile-processors`.

[201] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten. fTPM: A software-only implementation of a TPM chip. In *Proceedings of The 25th USENIX Security Symposium (UsenixSecurity'16)*, 2016.

[202] Raspberry PI. Model B+ on sale now at $35. `https://www.raspberrypi.org/blog/raspberry-pi-3-model-bplus-sale-now-35/`.

[203] Raspberry PI Foundation. Model B+. `https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/`.

[204] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in Android applications that feature anti-analysis techniques. In *Proceedings of 23rd Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[205] A. Reina, A. Fattori, F. Pagani, L. Cavallaro, and D. Bruschi. When hardware meets software: A bulletproof solution to forensic memory acquisition. In *Proceedings of the 28th Computer Security Applications Conference (ACSAC'12)*, 2012.

[206] R. Rivest. The MD5 message-digest algorithm. `https://www.ietf.org/rfc/rfc1321.txt`, 1992.

[207] D. Rosenberg. Reflections on trusting TrustZone. In *BlackHat USA*, 2014.

[208] X. Ruan. *Platform embedded security technology revealed: Safeguarding the future of computing with Intel embedded security and management engine*. Apress, 2014.

[209] J. Rutkowska. Intel x86 considered harmful. `http://blog.invisiblethings.org/papers/2015/x86_harmful.pdf`.

[210] J. Rutkowska and R. Wojtczuk. Preventing and detecting Xen hypervisor subversions. `http://www.invisiblethingslab.com/resources/bh08/part2-full.pdf`.

[211] I. Safonov and A. Matrosov. Excite project: all the truth about symbolic execution for BIOS security. http://2016.zeronights.org/program/9.

[212] Samsung. Artik. https://www.artik.io/.

[213] Samsung. Exynos processors. https://www.samsung.com/semiconductor/minisite/exynos/.

[214] M. A. M. P. Sanjeev Das, Jan Werner and F. Monrose. SoK: The challenges, pitfalls, and perils of using hardware performance counters for security. In *Proceedings of 40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[215] R. Sasmal. Fingerprint scanner: The ultimate security system. https://in.c.mi.com/thread-239612-1-0.html.

[216] Scaleway. Cloud service. https://www.scaleway.com/.

[217] J. Schiffman and D. Kaplan. The SMM rootkit revisited: Fun with USB. In *Proceedings of 9th International Conference on Availability, Reliability and Security (ARES'14)*, 2014.

[218] W. J. Schultz and H. A. Saladin. Electronic fuse for semiconductor devices, 1985. US Patent 4,562,454.

[219] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[220] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *Proceedings of 14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'17)*, 2017.

[221] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-shield: Enabling address space layout randomization for SGX programs. In *Proceedings of 24th Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[222] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS'07)*, 2007.

[223] D. Shen. Exploiting TrustZone on Android. In *Black Hat USA*, 2015.

[224] H. Shi, A. Alwabel, and J. Mirkovic. Cardinal pill testing of system virtual machines. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*, 2014.

[225] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 2016.

[226] W. Shi and S. Dustdar. The promise of edge computing. *IEEE Computer Magazine*, 2016.

[227] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV: Securing NFV states by using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, 2016.

[228] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of 24th Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[229] Shodan. Search engine for Internet-connected devices. `https://www.shodan.io/`, 2017.

[230] Siemens. M60 series ATC. `https://assets.new.siemens.com/siemens/assets/api/uuid:049fc482-9fa0-4fdb-bcdf-b72c54fe2534/version:`

1566316350/m60-atc-brochure.pdf, 2015.

[231] Siemens. SEPAC local controller software. https://assets.new.siemens.com/siemens/assets/api/uuid:69c0c411-a3bb-4158-a2ce-c4109d9daac2/version:1566316350/sepac-5-flyer.pdf, 2017.

[232] I. Skochinsky. Intel ME secrets: Hidden code in your chipset and how to discover what exactly it does. https://recon.cx/2014/slides/Recon%202014%20Skochinsky.pdf.

[233] S. Skorobogatov. Fault attacks on secure chips: From glitch to flash. https://www.cl.cam.ac.uk/~sps32/ECRYPT2011_1.pdf, 2011.

[234] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)*, 2008.

[235] D. Spencer. The adcanced transporation controller and applications for Oregon department of transportation. https://www.oregon.gov/ODOT/HWY/TRAFFIC-ROADWAY/docs/pdf/2013_conference/ATCforODOT.pdf, 2013.

[236] C. Spensky, H. Hu, and K. Leach. LO-PHI: Low-observable physical host instrumentation for malware analysis. In *Proceedings of 23rd Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[237] M. Spisak. Hardware-assisted rootkits: Abusing performance counters on the ARM and x86 architectures. In *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT'16)*, 2016.

[238] StatCounter. Mobile operating system market share worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide.

[239] Statcounter. Mobile vendor market share worldwide. http://gs.statcounter.com/vendor-market-share/mobile/worldwide.

[240] P. Stewin and I. Bystrov. Understanding DMA malware. In *Proceedings of 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'12)*, 2012.

[241] H. Sun, K. Sun, Y. Wang, and J. Jing. TrustOTP: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[242] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia. Trustdump: Reliable memory acquisition on smartphones. In *Proceedings of The 19th European Symposium on Research in Computer Security (ESORICS'14)*, 2014.

[243] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. TrustICE: Hardware-assisted isolated computing environments on mobile devices. In *Proceedings of The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'15)*, 2015.

[244] M. Sun, T. Wei, and J. Lui. TaintART: a practical multi-level information-flow tracking system for Android RunTime. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, 2016.

[245] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic reconstruction of Android malware behaviors. In *Proceedings of 22nd Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[246] A. Tang, S. Sethumadhavan, and S. Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*, 2017.

[247] Team Win. Team win recovery project. https://twrp.me/.

[248] Tencent Inc. TencentProtector. http://legu.qcloud.com/.

[249] A. Tereshkin and R. Wojtczuk. Introducing ring -3 rootkits. http://invisiblethingslab.com/itl/Resources.html.

[250] Texas. 2070 ATC equipment parts and accessories. http://esbd.cpa.state.tx.us/bid_show.cfm?bidid=139594, 2017.

[251] The GNU Multiple Precision Arithmetic Library. Pi with GMP. https://gmplib.org/.

[252] The MITRE Corporation. Common vulnerabilities and exposures. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=SSH, 2017.

[253] The Traffic Signal Museum. The traffic signal museum: Eagle signal model EF-15 traffic controller. http://www.trafficsignalmuseum.com/pages/ef15.html, 2015.

[254] The White House. Fact sheet: Announcing over $80 million in new federal investment and a soubling of participating communities in the White House amart cities initiative. https://obamawhitehouse.archives.gov/the-press-office/2016/09/26/fact-sheet-announcing-over-80-million-new-federal-investment-and, 2016.

[255] Ubuntu. sloccount. http://manpages.ubuntu.com/manpages/precise/man1/compute_all.1.html.

[256] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proceedings of 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[257] U.S. Department of Transportation. Traffic signal timing manual. `https://ops.fhwa.dot.gov/publications/fhwahop08024/index.htm`, 2008.

[258] T. Vidas and N. Christin. Evading Android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'14)*, 2014.

[259] N. Viennot, E. Garcia, and J. Nieh. A measurement study of Google Play. In *Proceedings of the ACM SIGMETRICS*, 2014.

[260] S. Vogl and C. Eckert. Using hardware performance events for instruction-level monitoring on the x86 architecture. In *Proceedings of the 2012 European Workshop on System Security (EuroSecâĂŹ12)*, 2012.

[261] Wandoujia. Wandoujia market. `http://www.wandoujia.com/apps`, 2017.

[262] J. Wang, K. Sun, and A. Stavrou. A dependability analysis of hardware-assisted polling integrity checking systems. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*, 2012.

[263] J. Wang, F. Zhang, K. Sun, and A. Stavrou. Firmware-assisted memory acquisition and analysis tools for digital forensic. In *Proceedings of the 6th International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE '11)*, 2011.

[264] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *Proceedings of 24th USENIX Security Symposium (USENIX Security'15)*, 2015.

[265] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, 2014.

[266] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *Proceedings of The 21st European Symposium on Research in Computer Security (ESORICS'16)*, 2016.

[267] C. Williams. Can't wait for ARM to power MOST of our cloud data centers. `https://www.theregister.co.uk/2017/03/09/microsoft_arm_server_followup/`.

[268] R. Wojtczuk and C. Kallenberg. Attacking UEFI boot script. `http://events.ccc.de/congress/2014/Fahrplan/system/attachments/2566/original/venamis_whitepaper.pdf`.

[269] R. Wojtczuk and J. Rutkowska. Attacking SMM memory via Intel CPU cache poisoning. `http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf`.

[270] M. Y. Wong and D. Lie. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In *Proceedings of 23nd Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[271] X. Wu, R. Dunne, Q. Zhang, and W. Shi. Edge computing enabled smart firefighting: opportunities and challenges. In *Proceedings of the 5th ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies*, 2017.

[272] Z. Wu. FPC1020 driver. `https://android.googlesource.com/kernel/msm/+/9f4561e8173cbc2d5a5cc0fcda3c0becf5ca9c74`.

[273] Xen project. Xen ARM with virtualization extensions. https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions.

[274] Xiaomi. Redmi 6. https://www.mi.com/global/redmi-6/.

[275] H. V. Xinyang Ge and T. Jaeger. SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *Proceedings of the 2014 Mobile Security Technologies (MoST'14)*, 2014.

[276] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[277] Yan, Lok Kwong and Yin, Heng. Droidscope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security'12)*, 2012.

[278] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu. AppSpear: Bytecode decrypting and DEX reassembling for packed Android malware. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'15)*, 2015.

[279] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: analyzing sensitive data transmission in Android for privacy leakage detection. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*, 2013.

[280] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the 2nd ACM/IEEE Symposium on Edge Computing (SEC'17)*, 2017.

[281] R. Yu. Android packers: facing the challenges, building solutions. In *Proceedings of the Virus Bulletin Conference (VB'14)*, 2014.

[282] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti, et al. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of 21st Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[283] D. Zhang. A set of code running on i.MX53 quick start board. `https://github.com/finallyjustice/imx53qsb-code`.

[284] F. Zhang. IOCheck: A framework to enhance the security of I/O devices at runtime. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, 2013.

[285] F. Zhang, K. Leach, A. Stavrou, and H. Wang. Using hardware features for increased debugging transparency. In *Proceedings of The 36th IEEE Symposium on Security and Privacy (S&P'15)*, pages 55–69, 2015.

[286] F. Zhang, K. Leach, K. Sun, and A. Stavrou. SPECTRE: A dependable introspection framework via system management mode. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, 2013.

[287] F. Zhang, K. Leach, H. Wang, and A. Stavrou. TrustLogin: Securing password-login on commodity operating systems. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'15)*, 2015.

[288] F. Zhang, H. Wang, K. Leach, and A. Stavrou. A framework to secure peripherals at runtime. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS'14)*, 2014.

[289] F. Zhang, J. Wang, K. Sun, and A. Stavrou. HyperCheck: A hardware-assisted integrity monitor. In *IEEE Transactions on Dependable and Secure Computing (TDSC'14)*, 2014.

[290] N. Zhang, K. Sun, W. Lou, and Y. T. Hou. Case: Cache-assisted secure execution on ARM processors. In *Proceedings of 37th IEEE Symposium on Security and Privacy (S&P'16)*, 2016.

[291] Q. Zhang, Z. Yu, W. Shi, and H. Zhong. Demo abstract: Evaps: Edge video analysis for public safety. In *Proceedings of the 1st IEEE/ACM Symposium on Edge Computing (SEC'16)*, 2016.

[292] X. Zhang, Y. Wang, and W. Shi. pcamp: Performance comparison of machine learning packages on the edges. In *Proceedings of USENIX Workshop on Hot Topics in Edge Computing (HotEdge'18)*, 2018.

[293] Y. Zhang, X. Luo, and H. Yin. DexHunter: Toward extracting hidden code from packed Android applications. In *Proceedings of The 20th European Symposium on Research in Computer Security (ESORICS'15).*, 2015.

[294] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in Android apps with permission use analysis. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*, 2013.

[295] Zheng, Min and Sun, Mingshen and Lui, John CS. DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC'14)*, 2014.

ABSTRACT

SECURING ARM PLATFORM: FROM SOFTWARE-BASED TO HARDWARE-BASED APPROACHES

by

ZHENYU NING

May 2020

Advisor: Dr. Fengwei Zhang

Major: Computer Science

Degree: Doctor of Philosophy

With the rapid proliferation of the ARM architecture on smart mobile phones and Internet of Things (IoT) devices, the security of ARM platform becomes an emerging problem. In recent years, the number of malware identified on ARM platforms, especially on Android, shows explosive growth. Evasion techniques are also used in these malware to escape from being detected by existing analysis systems.

In our research, we first present a software-based mechanism to increase the accuracy of existing static analysis tools by reassembleable bytecode extraction. Our solution collects bytecode and data at runtime, and then reassemble them offline to help static analysis tools to reveal the hidden behavior in an application.

Further, we implement a hardware-based transparent malware analysis framework for general ARM platforms to defend against the traditional evasion techniques. Our framework leverages hardware debugging features and Trusted Execution Environment (TEE) to achieve transparent tracing and debugging with reasonable overhead.

To learn the security of the involved hardware debugging features, we perform a com-

prehensive study on the ARM debugging features and summarize the security implications. Based on the implications, we design a novel attack scenario that achieves privilege escalation via misusing the debugging features in inter-processor debugging model.

The attack has raised our concern on the security of TEEs and Cyber-physical System (CPS). For a better understanding of the security of TEEs, we investigate the security of various TEEs on different architectures and platforms, and state the security challenges. A study of the deploying the TEEs on edge platform is also presented. For the security of the CPS, we conduct an analysis on the real-world traffic signal infrastructure and summarize the security problems.

AUTOBIOGRAPHICAL STATEMENT

ZHENYU NING

Zhenyu Ning received his M.S. degree and B.S. in Computer Science from Tongji University in 2011 and 2008, respectively. Before starting the Ph.D. program, he had spent more than 4 years in industry as software engineer. His current research focuses on different areas of security and privacy, including system security, mobile security, IoT security, transportation security, trusted execution environment, and hardware-assisted security.