Revisiting ARM Debugging Features: Nailgun and its Defense

Zhenyu Ning, Chenxu Wang, Yinhua Chen, Fengwei Zhang, and Jiannong Cao

Abstract—Processors nowadays are consistently equipped with debugging features to facilitate 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. Meanwhile, 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 article, we perform a comprehensive security analysis of the ARM debugging features and summarize the security implications. To understand the impact of the implications, we also investigate a series of platforms with ARM-A architecture in different product domains (i.e., development boards, IoT devices, cloud servers, and mobile devices). We consider that the analysis and investigation expose a new attacking surface that universally exists in platforms with ARM-A architecture. 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. Our analysis also indicates that ARM-R and ARM-M platforms may suffer from the same issue. To defend against the attack, we discuss potential mitigations from different perspectives in the ARM ecosystem. Finally, a practical defense mechanism based on ARM virtualization technology is presented, and the evaluation result shows that our defense can prevent NAILGUN with a negligible performance penalty.

Index Terms—ARM debugging architecture, trusted execution environment, privilege escalation, virtualization

1 Introduction

Most of the processors today utilize a debugging architecture to facilitate on-chip debugging. For example, the x86 architecture provides six debug registers to support hardware breakpoints and debug exceptions [1], and the Intel Processor Trace [2] is a hardware-assisted debugging feature that garners attention in recent research [3], [4]. 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 [5], [6]. Moreover, ARM also introduces hardware components, such as the Embedded Trace Macrocell [7] 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 ports is the Joint Test Action Group (JTAG) port defined by IEEE Standard 1149.1 [9], 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 [10], ARM DS-5 [11], and OpenOCD [12]), developers can access the memory and registers of the target efficiently and conveniently.

To authorize external debugging tools in different usage scenarios, ARM designs several authentication signals. Specifically, four debug authentication signals control whether non-invasive debugging or invasive debugging (see Section 2.2) is prohibited when a target processor is in a 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 purposes. 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 [5]. As shown in the left side of Fig. 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.

- *Zhenyu Ning, Yinhua Chen, and Fengwei Zhang are with the Southern University of Science and Technology, Shenzhen 518055, China. E-mail: {ningzy, zhangfw}@sustech.edu.cn, 11612127@mail.sustech.edu.cn.*
- *Chenxu Wang is with the Southern University of Science and Technology, Shenzhen 518055, China, and also with the Hong Kong Polytechnic University, Hong Kong. E-mail: 11711715@mail.sustech.edu.cn.*
- *Jiannong Cao is with the Hong Kong Polytechnic University, Hong Kong. E-mail: csjcao@comp.polyu.edu.hk.*

Manuscript received 18 December 2020; revised 22 November 2021; accepted 22 December 2021. Date of publication 31 December 2021; date of current version 16 January 2023.
This work was supported in part by the National Natural Science Foundation of China under Grants 62102175 and 62002151, and Science Technology and Innovation Commission of Shenzhen Municipality under Grant SGDX20201103095408029.
(Corresponding author: Fengwei Zhang.)
Digital Object Identifier no. 10.1109/TDSC.2021.3139840

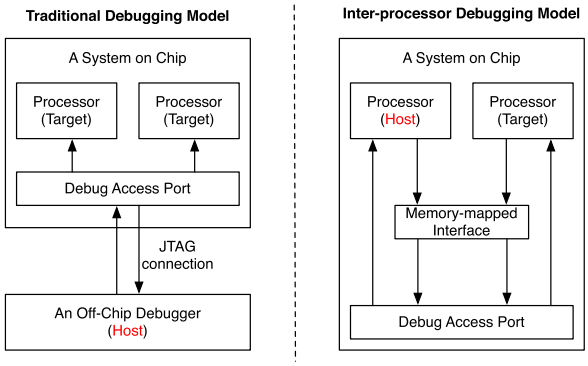


Fig. 1. Debug models in ARM architecture.

The right side of Fig. 1 presents the new debugging model introduced since ARMv7. In this model, a memory-mapped interface is used to map a group of 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 a privilege escalation via misusing the debugging features. With the help of another processor, a low-privilege processor can obtain arbitrary access to high-privilege resources 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 hypervisor-level privilege and the secure privilege levels provided by TrustZone [13].

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-A platforms. Specifically, we investigate the default status and the management mechanism of these signals on 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.

Our investigation finds that the debug authentication signals are fully or partially enabled on the investigated platforms. Moreover, 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 a 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 indicates that NAILGUN can leak the fingerprint image stored in TrustZone from 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 6.*

Our findings have been reported to the related hardware manufacturers including IoT device vendors such as Raspberry PI Foundation [14], commercial cloud providers such as miniNode [15], Packet [16], Scaleway [17], and mobile device vendors such as Motorola [18], Samsung [19], Huawei [20], Xiaomi [21]. Meanwhile, SoC manufacturers are notified by their customers (e.g., the mobile device vendors) and work with us for a practical solution. We have also informed ARM about the security implications.

Moreover, we design a practical defense of NAILGUN based on ARM virtualization technology. Specifically, we protect the debug registers with additional memory address translation layer in a higher privilege level. To evaluate our design, we implement a prototype of the defense mechanism on the Raspberry PI 3 Model B+ board. The experiments show that our defense can prevent NAILGUN with a negligible performance penalty.

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 in different ARM architectures including ARM-A, ARM-R, and ARM-M to acquire a comprehensive understanding of the debugging features, and summarize

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

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-A architectures and platforms including 64-bit ARMv8-A Juno Board, 32-bit ARMv8-A Raspberry PI 3 Model B+, and ARMv7-A Huawei Mate 7. To extend the breadth of the attack, we design different attacking scenarios based on both non-invasive and invasive debugging features. The experiments 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 comprehensive countermeasures to our attacks from different perspectives in the ARM ecosystem. Moreover, we design a practical defense mechanism based on ARM virtualization technology and implement a prototype of the defense on a Raspberry PI 3 Model B+ board. The evaluation results show that our defense can prevent NAILGUN with a negligible performance penalty.

This paper is an extended version of our previous work [22] published in IEEE Symposium on Security & Privacy 2019. Based on that work, we design and implement a practical defense mechanism to prevent NAILGUN attack with a negligible performance overhead. We also extend previous attack scenarios to show NAILGUN can be used to access privileged registers from a low-privilege mode. A comprehensive analysis of related vulnerability on ARM-R and ARM-M architectures is also presented. The main differences between these two versions are listed below:

- Based on ARM virtualization technology, we design a practical defense mechanism of NAILGUN by enabling an additional memory translation layer to restrict the access to debug registers. With this restriction, we prevent NAILGUN by disabling the attacker's ability of forcing the processors to enter a high-privilege mode in debug state.
- We implement a prototype of the proposed defense on a 32-bit Raspberry PI 3 Model B+ board. The evaluation shows that the defense mechanism is capable of preventing NAILGUN with less than 1.1% performance overhead.
- We extend the attack scenario of NAILGUN and show that NAILGUN can also be used to access privileged registers from a low-privilege mode.
- We perform a comprehensive analysis of the vulnerabilities exploited by NAILGUNon ARM-R and ARM-M architectures. Our analysis shows that these architectures may also suffer from NAILGUN attack.

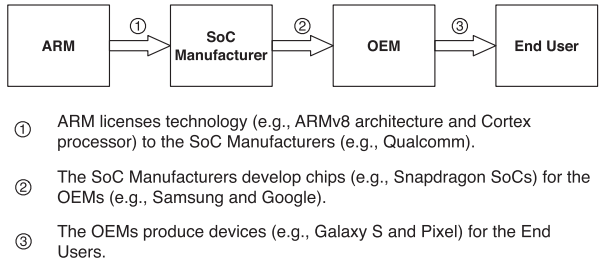


Fig. 2. Relationships in the ARM ecosystem.

- Moreover, we will release the source code of the designed defense mechanism and representative attack scenarios with detailed instructions to reproduce the attack in [23].

2 BACKGROUND

2.1 ARM, SoC Manufacturer, and OEM

Fig. 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 smartphones 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 the role of the SoC manufacturer and develops the Exynos SoCs.

2.2 ARM Debugging Architecture

The ARM architecture defines both invasive and non-invasive debugging features [5], [6]. 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. In contrast, the debugging features such as tracing (via the Embedded Trace Macrocell) and monitoring (via the Performance Monitor Unit) are non-invasive debugging.

2.3 ARM Security Extension

The ARM Security Extensions [13], known as TrustZone technology, allows the processor to run in secure and non-secure states. The memory is 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-A architecture [6], 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.

2.4 ARM Debug Authentication Signals

ARMv7-A and ARMv8-A architectures define four signals for external debug authentication, i.e., DBGEN, NIDEN, SPIDEN, and SPNIDEN. The DBGEN signal controls whether the non-secure invasive debugging is allowed in the system. While the signals DBGEN or NIDEN is high, the non-secure non-invasive debugging is enabled. Similarly, the SPIDEN and SPNIDEN signals control the secure invasive and non-invasive debugging, 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.

Compared with ARM-A architectures, ARM-R and ARM-M provide different signals for debug authentication. Since ARMv7-R and ARMv7-M lack the support Security Extension, they only define the DBGEN and NIDEN signals to control the invasive and non-invasive debugging, respectively. ARMv8-R supports Virtualization Extension and defines two additional signals (i.e., HIDEN and HNIDEN) for EL2. The HIDEN signal controls the invasive debugging, while the HNIDEN controls the non-invasive debugging. Moreover, since ARMv8-M supports Security Extension, the debug authentication signals in ARMv8-R are identical to those in ARM-A architecture.

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 this 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.

2.5 ARM CoreSight Architecture

The ARM CoreSight architecture [24] 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) [7] 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.

2.6 ARMv8-A Memory Address Translation

For memory management, ARMv8-A architecture introduces three types of address, including the virtual address (VA), the intermediate physical address (IPA), and the physical address (PA). This architecture also provides a procedure of mapping one address type to another (i.e., the address translation). In a typical process of address translation, the Memory Management Unit (MMU) takes the input address of one type (e.g., the VA), performs the corresponding translation, then returns an output address of another type (e.g., the IPA). If the MMU fails to translate the input address, it generates an MMU fault with a specific reason (e.g., the translation fault caused by an invalid page in the translation table).

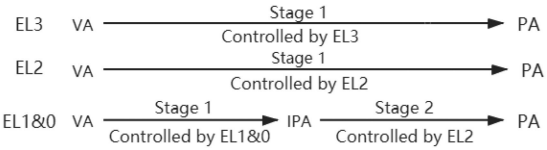


Fig. 3. ARMv8-A address translation.

Fig. 3 shows three types of translation regimes defined by ARMv8-A architecture. Each exception level has its own Stage 1 translation, and an additional translation layer (i.e., Stage 2 translation) controlled by EL2 is introduced in EL1&0. The additional translation is generally utilized for hypervisors to manage guest memory, and it can also be used for enhancing kernel security [25].

3 Security Implications of the Debugging Architecture

In this section, we carefully investigate the non-invasive and invasive debugging mechanisms documented in the Technique Reference Manuals (TRM) [5], [6] of ARM-A architectures, and reveal the vulnerability and security implications indicated by the manual. The differences between ARM-A architectures and other ARM architectures are presented in Section 5.2.6. *Note that we assume the required debug authentication signals are enabled in this section, and this assumption is proved to be reasonable and practical in Section 4.*

3.1 Non-Invasive Debugging

The non-invasive debugging does not allow halting a processor and introspecting 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-A architecture, the PMU is controlled by a group of registers 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 the help of the debug authentication signals. The TPM bit of the MDCR register is introduced in ARMv8-A 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 [6].

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

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

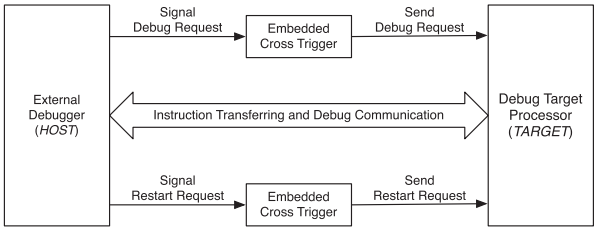


Fig. 4. Invasive debugging model.

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 [6].

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.

3.2 Invasive Debugging

The invasive debugging allows an external debugger to halt a target processor and access the resources on the processor via the debugging architecture. Fig. 4 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 3.2.2) 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-A. 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 in the inter-processor debugging model. In this section, we discuss the security implications of the inter-processor debugging under the legacy debug authentication mechanism.

3.2.1 Entering and Existing Debug State

To achieve the invasive debugging in the TARGET, one should make the TARGET run in the debug state. 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 causes the processor to halt and enter the debug state directly. A more general approach to enter the debug state is to send an external debug request via the ECT. Each processor in a multi-processor system is embedded with a separated CTI (i.e., interface to ECT), and the memory-mapped interface makes the CTI on a processor available to other processors. Thus, the HOST can leverage the CTI of the TARGET to send the external debug request and make the TARGET enter the debug state. Similarly, a restart request can be used to exit the debug state.

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

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

3.2.2 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.

To enable data transferring between a HOST in the normal state and a TARGET in the debug state, Debug Communication Channel (DCC) is introduced. In ARMv8-A architecture, three registers exist in the DCC. A 32-bit DBGDTRTX register is used to transfer data from the TARGET to the HOST, while a 32-bit DBGDTRRX register is used to receive data from the HOST. Moreover, a 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 can 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 consider the privilege of the HOST.

3.2.3 Privilege Escalation

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.

TABLE 1
Debug Authentication Signals on Real Devices

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

The 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.

3.3 Summary

Both the non-invasive and invasive debugging 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 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.

Moreover, the discussed implications are difficult to be identified by automatic analysis or test since they require a comprehensive understanding of the debugging architecture. However, once they are discovered, we can leverage automatic analysis to verify whether a specific device is vulnerable or not. For example, the value of Debug Authentication Status Register (`DBGAUTHSTATUS`) indicates the status of debug authentication signals in a device, and analyzers can use this status to evaluate whether the device is vulnerable to NAILGUN attach in large-scale automatic analysis.

4 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 investigate 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 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 depend on the SoC manufacturers and the OEMs, we select various devices powered by different SoCs and OEMs as the investigation target.

To understand the status of debug authentication signals, we survey the devices applied in different product domains including development boards (e.g., the official Juno Board [26] released by ARM), Internet of Things (IoT) devices (e.g., the Raspberry PI 3 [14]), commercial cloud platforms (e.g., miniNodes [15], Packet [16], and Scaleway [17]), and mobile devices (e.g., Google Nexus 6, Samsung Galaxy Note 2, Huawei Mate 7, Motorola E4 Plus, and Xiaomi Redmi 6). For these devices, we read the status of debug authentication signals via the Debug Authentication Status Register (`DBGAUTHSTATUS`), and the result is shown in Table 1. According to our investigation, these signals are fully or partially enabled on all the tested devices by default, making them vulnerable to the aforementioned isolation violation and privilege escalation. The detailed result of the survey can be found in [22].

Furthermore, to learn the deployed signal management mechanism, we collect information from the publicly available manuals and the source code released by the hardware vendors. Based on our survey, there is no publicly available management mechanism for these signals on all tested devices except for development boards. The documented management mechanism of development boards is either incomplete or not fully functional. On the one hand, the unavailable management mechanism may help to prevent malicious access to the debug authentication signals. On the other hand, it stops the user from disabling the debug authentication signals for defense purposes. The detailed analysis result can be found in [22].

5 NAILGUN ATTACK

To verify the security implications concluded in Section 3 and the findings of the debug authentication signals described in Section 4, we craft an attack named NAILGUN and implement it in several different ARM-A 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 one attacking scenarios for non-invasive debugging and three attacking scenarios for invasive debugging, respectively. With the non-invasive debugging feature, NAILGUN is able to infer AES encryption keys, which are isolated in EL3, via executing an application in non-secure EL1. Moreover, we craft a prototype of NAILGUN attack to read the privileged resource via invasive debugging attack in Raspberry PI 3 Model B+. To further leverage the invasive debugging mechanism, we demonstrate that an application running in non-secure EL1 can execute arbitrary payloads in EL3 with NAILGUN. 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. We also offer the difference of NAILGUN attack between 64-bit ARMv8-A and other ARM-A architectures. Furthermore, the analysis on ARM-R and ARM-M debugging architectures is presented to explore NAILGUN attack on these architectures.

5.1 Threat Model and Assumptions

In our attack, we make no assumption about the version or type of the operating 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 the access to the CoreSight components and debug registers, which are typically mapped to some physical memory regions in the system. Note that it normally requires non-secure EL1 privilege to map the CoreSight components and debug registers to the virtual memory address space.

5.2 Attack Scenarios

5.2.1 *Inferring Encryption Keys via Non-Invasive Debugging*

AES algorithm has been proved to be vulnerable to side-channel attacks [27], [28], [29], [30], [31], [32] since the table-lookup based implementation leaks the information about the encryption key. With the addresses of the accessed table entries, an attacker can efficiently rebuild the encryption key. In this attack, we assume a secure application running in TrustZone provides AES encryption service with a predefined encryption key. The secure application also exposes an interface to the non-secure OS for encrypting 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 invoking the encryption interface from the non-secure OS.

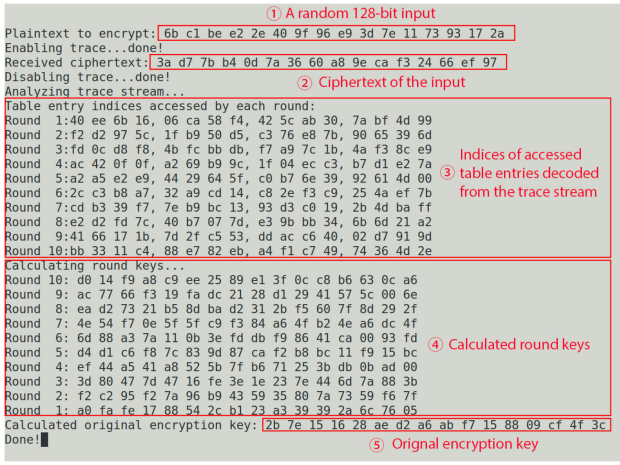


Fig. 5. Retrieving the AES encryption key.

The violation of privilege isolation described in Section 3.1 enables a non-secure application to learn the information about the secure execution. Specifically, we leverage the ETM instruction trace to rebuild the addresses of executed instructions and data-address trace to record the data involved in data processing instructions (e.g., `ldr`, `str`, `mov`, and etc.). With this information, it is trivial to learn the addresses of the instructions performing AES table lookup and the memory addresses of the accessed table entries. Finally, we infer the AES encryption key with the recorded addresses.

To demonstrate the attack, we initially build a bare-metal environment on an NXP i.MX53 Quick Start Board [33]. The board is integrated with a single Cortex-A8 processor enabling the data-address trace, and we build our environment based on an open-source project [34] 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 [35] to the environment and run it in the secure state with a predefined 128-bit key stored in the secure memory. A non-secure application can provide a plaintext and send an encryption request via the interface exposed by the secure state.

Fig. 5 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 accessed table entries in each encryption round and convert them into entry indices by base addresses of the tables, as shown in ③. With the indices and the ciphertext, we reverse the encryption algorithm and calculate the round keys in ④. Finally, NAILGUN decodes the original encryption key in ⑤ via the round key and accessed table entries in round 1.

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

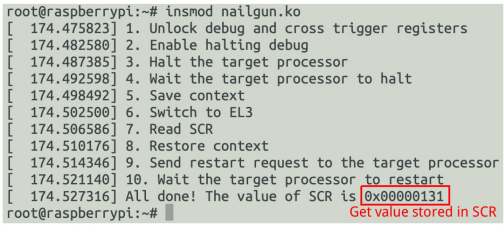


Fig. 6. Reading the secure configuration register.

5.2.2 Privileged Resource Access via Invasive Debugging

The invasive debugging is more powerful than non-invasive debugging since we can halt the target processor and access the restricted resources via the debugging architecture. In this section, we design and implement an attack scenario to show that NAILGUN can be used to access privileged resources via privilege escalation on Raspberry PI 3 Model B+ board. The board contains a quad-core Cortex-A53 cluster, and the official firmware makes the Cortex-A53 on Raspberry PI run in the 32-bit ARMv8 mode.

In this attack scenario, we craft the NAILGUN attack to read the Secure Configuration Register (SCR) from non-secure EL1, which violates the access restriction that this register is supposed to be accessed only in EL3. Specifically, we leverage one of the Cortex-A53 processors (HOST) in the cluster to debug and control another Cortex-A53 processor (TARGET) in the same cluster via a Loadable Kernel Module (LKM) running within the Linux kernel. With the debugging architecture, we first halt the TARGET from the HOST and promote the privilege of the HOST to EL3. Next, the TARGET is instructed to access the SCR and return the result to the HOST. Finally, we resume the TARGET to normal execution. Fig. 6 indicates the attack process, and the right-bottom rectangle shows the value we read from SCR.

5.2.3 Arbitrary Payload Execution via Invasive Debugging

In this section, we further extend the aforementioned attack and leverage the invasive debugging to achieve arbitrary payload execution.

The EDITR register enables an attacker to execute instructions on the TARGET from the HOST. However, not all instructions can be executed via the EDITR register. For example, executing branch instructions (e.g., b, bl, and blr instructions) in EDITR leads to an unpredictable result. However, a malicious payload in real-world normally contains branch instructions. To bypass this restriction, NAILGUN crafts a robust solution to execute an 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, we overwrite the instruction pointed by this register to redirect the control flow of the non-debug state to our payload.

The second requirement cannot be satisfied by simply executing dcps instructions since the dcps instructions only changes the exception level of a processor in the debug state. The exception level of the processor is reverted while exiting the debug state. We note that the smc instruction in the non-debug state asserts a Secure Monitor Call (SMC) exception, which takes the processor to the corresponding exception handler in EL3. Thus, we manipulate the exception vector stored in the VBAR_EL3 register and leverage this instruction to enter the malicious payload in EL3.

The third requirement is also critical since NAILGUN modifies the memory pointed by the DLR_EL0 and VBAR_EL3 registers to meet the previous requirements. To avoid the side-effect introduced by the manipulation, NAILGUN needs to rollback these changes in the TARGET after executing the payload. Moreover, NAILGUN needs to store the context at the very beginning of the payload and revert it at the end of the payload.

We implement NAILGUN on a 64-bit ARMv8-A Juno r1 board [26] to show that the *Implications 2-4* lead to arbitrary payload execution in EL3. The board contains two Cortex-A57 processors and four Cortex-A53 processors, and we use ARM Trusted Firmware (ATF) [36] and Linaro's deliverables on OpenEmbedded Linux for Juno [37] 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.

Fig. 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 Fig. 7a, the TARGET is halted by the HOST before the execution of the mov instruction, and 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 handler for SMC exception locates at offset 0x400 of the exception vector [6]. Fig. 7b shows the memory of the TARGET before exiting the debug state. In 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 corresponding 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 with the pc register pointing to the malicious smc instruction, as shown in Fig. 7c. The execution of the smc instruction takes the TARGET to the status shown in Fig. 7d. The TARGET is handling the SMC exception; the corresponding exception return address (pointed by the ELR_EL3 register) is the address of the instruction next to the executed smc instruction. Our manipulation of the exception handler leads to the execution of the payload, which performs malicious activities and restores the changed memory. At the end of the payload, an eret instruction is leveraged to switch back to

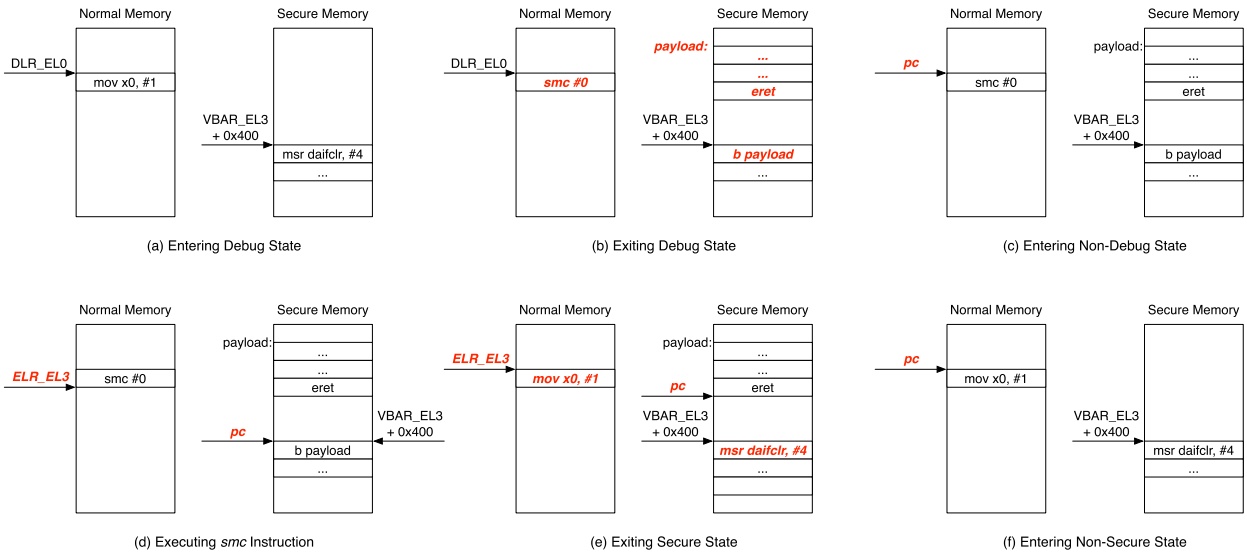


Fig. 7. Executing arbitrary payload in the secure state.

the non-secure state. Fig. 7e indicates the memory and status before the switch; 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 Fig. 7f, the `TARGET` enters the non-secure state again, and the memory and status look the same as that in Fig. 7a.

Fig. 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 that the attack has succeeded, we also extract the current exception level from the `CurrentEL` register. The last line of the outputs in Fig. 8 indicates that NAILGUN is able to execute arbitrary code in EL3, which owns the highest privilege over the whole system.

5.2.4 *Fingerprint Extraction in a Real-World Mobile Phone*

To learn the impact of NAILGUN on real-world devices, we also show that NAILGUN is able to leak sensitive information stored in secure memory. Currently, one of the frequently used security features in mobile phones is fingerprint authentication [21], [38], [39], and the OEMs store the fingerprint image in TrustZone to enhance the security of the device [40], [41], [42]. In this experiment, we use Huawei Mate 7 [38] to demonstrate that the fingerprint image can be extracted by an LKM running in the non-secure EL1 with the help of NAILGUN. The Huawei Mate 7 is powered by HiSilicon Kirin 925 SoC, which integrates a quad-core Cortex-A15 cluster and a quad-core Cortex-A7 cluster. The FPC1020 [43] fingerprint sensor is used in Mate 7 to capture the fingerprint image. This phone is selected since the product specification [44] and driver source code [45] of FPC1020 are publicly available, which reduces the engineering effort of implementing the attack.

```
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
Exception Level read from the payload
```

Fig. 8. Executing payload in TrustZone via an LKM.

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.

The format of the image data is well-documented in the FPC1020 product specification [44]. According to the specification, each byte of the data indicates the grayscale level of a single pixel. Thus, with the extracted image data, it is trivial to craft a grayscale fingerprint image. Fig. 9 shows the fingerprint image extracted from Huawei Mate 7 via NAILGUN. It demonstrates that NAILGUN is able to leak sensitive data from the TEE in commercial mobile phones with some engineering efforts. Note that the implementation of NAILGUN in Huawei Mate 7 is different from that in ARM Juno board due to architecture differences (see Section 5.2.5).

5.2.5 *NAILGUN in Other ARM-A Architectures*

Section 3 discusses the security implications of 64-bit ARMv8-A debugging architecture, and similar implications

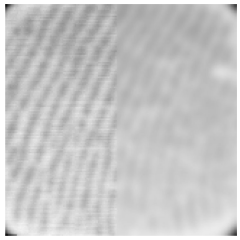


Fig. 9. Fingerprint Image Leaked by NAILGUN from Huawei Mate 7. The right half of the image is blurred for privacy concerns.

exist in 32-bit ARMv8-A and ARMv7-A architecture. However, there are also some major differences in implementing NAILGUN in these architectures, and we discuss them in this section. Moreover, a preliminary study of NAILGUN on ARMv9-A architecture is presented.

32-bit ARMv8-A Architecture. We implement prototypes of NAILGUN with 32-bit ARMv8-A 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-A architecture, and the major difference is the usage of the `EDITR`. In the 64-bit ARMv8-A, 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 mode. For example, the binary representation of the `dcps3` instruction is `0xD4A00003` and `0xF78F8003` in 32-bit and 64-bit ARMv8-A, respectively. In the 64-bit ARMv8-A architecture, a processor in the debug state executes 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-A architecture.

ARMv7-A Architecture. In regard to Armv7-A, we implement NAILGUN on Huawei Mate 7 as discussed in Section 5.2.4, and there are three major differences between NAILGUN on ARMv7-A and ARMv8-A architectures. First, the ECT is not required to halt and restart a processor in ARMv7-A. Writing 1 to the bit[0] and bit[1] of the Debug Run Control Register (DBGDRCR) can directly halt and restart a processor, respectively. Second, the `ITRen` bit of the `EDSCR` controls whether the `EDITR` is enabled in ARMv7-A 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-A 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.

ARMv9-A Architecture. Since the commercial ARMv9-A device is not available yet, we implement a prototype of NAILGUN on Fixed Virtual Platforms (FVP) [46] with ARMv9-A Realm Management Extension [47]. The ARMv9-A architecture splits the execution environment into four states, i.e., non-secure state (non-secure EL0-EL2 in ARMv8-A), secure state (secure EL0-EL2 in ARMv8-A), root state (EL3 in ARMv8-A), and realm state (newly introduced in ARMv9-A). Two additional debug authentication signals `RLPIDEN` and `RTPIDEN`are introduced to manage the debugging in the realm and root state, respectively. Since the debugging in the root state and realm state is disabled by default on the FVP, we are not able to evaluate NAILGUN's ability to attack the corresponding state. Moreover, without entering root state in debug state, the ability of NAILGUN to attack secure state is also restricted. However, we succeed in accessing non-secure EL2 resources from non-secure EL1 with NAILGUN, which illustrates that the implications concluded in Section 3 may still exist in ARMv9-A architecture.

5.2.6 NAILGUN in ARM-R and ARM-M Architectures

To learn the impact of NAILGUN in ARM-R and ARM-M architecture, we perform an extensive analysis to their debugging architectures on the vulnerabilities exploited by NAILGUN. However, due to the availability of devices meeting the requirements of NAILGUN, we focus on the analysis of architecture reference manuals, and the implementation of NAILGUN on real-world devices is considered as our further work.

ARMv7-R Architecture. The privilege modes and debugging architecture in ARMv7-R [5] architecture are similar to those in ARMv7-A architecture. Nevertheless, since the Virtualization Extension and Security Extension are not supported in ARMv7-R architecture, the `Hyp` mode (EL2 in ARMv8-A) and `Monitor`mode (EL3 in ARMv8-A) are not present. Thus, the PL1 (EL1 in ARMv8-A) owns the highest privilege in the architecture.

Non-invasive and invasive debugging attack in NAILGUN requires the access to ETM registers and debug registers, respectively. In platforms with ARM-A architecture, it normally requires EL1 privilege to map the physical addresses of these registers to virtual addresses before accessing them. However, the mapping is not required in ARMv7-R architecture. Instead, Memory Protection Unit (MPU) is proposed, and the memory access is restricted by protection regions. Therefore, if an MPU configuration allows PL0 (EL0 in ARMv8-A) to access these registers, NAILGUN may be leveraged to achieve the privilege escalation from PL0 to PL1. Otherwise, it requires the highest privilege (PL1) in the system to access these registers, and no privilege escalation is expected.

ARMv8-R Architecture. The ARMv8-R architecture [48], [49] follows the design of ARMv8-A architecture, but the Security Extension is still absent. To support Virtualization Extension, an additional MPU is introduced in EL2 to achieve the hypervisor-level memory access control. In regard to the debug authentication signals, two additional signals `HIDEN` and `HNIDEN` are introduced to control the invasive debugging and non-invasive debugging in EL2, respectively. To the best of our knowledge, the debug authentication mechanism in ARMv8-R architecture still lacks the consideration of the privilege in the `HOST`. Thus, we consider it may suffer from NAILGUN if the configuration of MPU allows the access to the ETM or debug registers from EL0 or EL1.

ARMv7-M Architecture. Unlike the ARMv7-A and ARMv7-R architecture, the ARMv7-M architecture only defines two processor modes: Thread Mode and Handler Mode. Thread Mode is designed for applications, while Handler Mode is used for handling exceptions. The execution in Handler Mode is always considered as privileged. However, execution in Thread Mode can be either privileged or unprivileged, and the `nPRIV` bit of `CONTROL` register indicates the current privilege status of Thread Mode. The system memory model in ARMv7-M also allows the MPU to manage

access permissions to various memory regions under different processor modes.

According to the reference manual [50], the debug registers in ARMv7-M architecture are located at a fixed memory address ranging from `0xE000EDF0` to `0xE000EEFF`. However, how the address conflict is solved in a multiprocessor system is not clarified in the manual. Moreover, the corresponding debugging architecture does not involve the instruction transfer channels, and `HOST` is not able to make `TARGET` execute instructions in debug state. Thus, we consider the invasive debugging attack would not work in ARMv7-M architecture.

The non-invasive debugging attack in NAILGUN is based on non-invasive tracing hardware ETM. However, ETM is not a mandatory component for ARMv7-M devices due to the special deployment scenarios. Moreover, the access to the ETM registers relies on the configuration of MPU in a device with ETM support. Thus, the non-invasive debugging attack in ARMv7-M architecture requires a platform with ETM support and an MPU configuration that allows unprivileged access to ETM registers.

ARMv8-M Architecture. Based on ARMv7-M architecture, ARMv8-M architecture [51] adds support to the Security Extension. Similar to the Security Extension in ARM-A architecture, the memory space and processor state are divided into secure and non-secure ones. The transition from the non-secure state to the secure state is achieved by Secure Gateway (`SG`) instruction, while the interstating branch instructions (`BXNS` and `BLXNS`) targeting a non-secure memory address bring the processor back to the non-secure state. Moreover, the debug authentication signal could be overwritten by software running in The secure state via the Debug Authentication Control Register (`DAUTHCTRL`).

The invasive debugging attack in NAILGUN requires a `HOST` processor and a `TARGET` processor in the same SoC. However, most available ARMv8-M devices contain a single Cortex-M processor [52], [53], [54], [55], [56], [57]. Although some devices [58], [59] claim there are two Cortex-M processors in the SoC, one of them is acting as a coprocessor. Thus, we consider launching an invasive debugging attack on ARMv8-M devices would be challenging due to the lack of multi-processor support.

Similar to ARMv7-M architecture, ETM is an optional component for ARMv8-M devices, and the access to ETM registers is restricted by MPU. Therefore, the non-invasive debugging attack in ARMv8-M devices would be possible if the MPU is configured inappropriately.

Nevertheless, since ARMv8-M architecture allows the software to overwrite the debug authentication signals, the SoC manufacturers and OEMs can tackle the attack with software patches.

6 COUNTERMEASURE

In this section, we focus on the countermeasures in ARM-A architecture as devices with ARM-A architecture are the main victims of NAILGUN attack.

6.1 Disabling the Signals?

Since the feasibility of NAILGUN attack depends on enabled debug authentication signals, an intuitive defense is to disable these signals. However, based on the ARM-A Architecture Reference Manual [5], [6], the analysis in Section 4, 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 debug authentication signals. The invasive and non-invasive debugging features are heavily used to build analysis systems [60], [61], [62], [63], [64], [65], [66], [67], [68], [69]. Disabling the debug authentication signals would directly make these systems fully or partially malfunction. In the ARMv7-A architecture [5], the situation is even worse since the functionality of the widely used Performance Monitor Unit (PMU) [68], [70], [71], [72], [73], [74], [75] 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 4, 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 attack 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 [76].

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. Even if the vendors deploy such restrictions (e.g., disabling the debug authentication signals) on future devices, it still introduces a considerable cost for the callback and maintenance process of current devices.

6.2 Comprehensive Countermeasure

We consider NAILGUN attack is caused by two reasons: 1) the debug authentication signals defined by ARM does not fully

consider the scenario of inter-processor debugging, which leads to the security implications described in Section 3; 2) the configuration of debug authentication signals and the management mechanism 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. In general, we suggest a comprehensive defense across different roles in the ARM ecosystem. As a supplement, we also provide a practical defense against NAILGUN attack, which restricts the access to the debug registers.

6.2.1 Defense From ARM

Implementing additional restrictions in the inter-processor debugging model. The key issue that drives the existence of NAILGUN attack 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, redesigning them and making 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 is required. For example, in the invasive debugging, the `TARGET` should check the privilege of the `HOST` and respond 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. For usability concerns, the high-privilege debugger can provide interfaces with additional authentication mechanisms to help a low-privilege debugger to debug a high-privilege processor.

Refining the granularity of the debug authentication signals. Other than distinguishing the on-chip and off-chip `HOST`, we also suggest that 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.

6.2.2 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. Precisely, 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 [26], 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 state. We consider this restriction method too strict since it somehow restricts the legitimate usage of the debugging features. It could be improved by allowing the highest privilege level in the non-secure state (non-secure EL2) to manage the signals related to the non-secure debugging. The design of the i.MX53 SoC [77], 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 without constraints. We consider this restriction method too loose since it introduces a potential attack surface to these signals. It could be improved by adopting additional security mechanisms before allowing the external debugger to change the authentication signals.

Applying access control to the debug registers. NAILGUN attack relies on the access to the debug registers, which is typically achieved by memory-mapped interfaces. Intuitively, restrictions on accessing these registers would help to enhance the security of the platform. During the responsible disclosure to MediaTek, we learn that they have the hardware-based technology for the TrustZone boundary division, and they are planning to use it to restrict the access to the debug registers to mitigate the reported attack. The ARMv8.4-A architecture also introduces an extension to restrict the non-secure access to debugging registers. However, simply using the secure/non-secure state to apply the access control would still lead to privilege escalation in a single state. For example, our study on ARMv9 FVP in Section 5.2.5 shows that it is possible to access non-secure EL2 resources from non-secure EL1 with NAILGUN.

6.2.3 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 helps to protect the secure content from the non-secure state and 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 privileged applications is allowed, while the misuse from unprivileged applications is forbidden. Similarly, applying this restriction to the access of CoreSight components and debug registers can also form an effective defense since the debugging features are exploited via the CoreSight components and the debug registers.

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 the access to these registers. In the Linux kernel, the common 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 stock kernel (many android devices have adopted this strategy). In this case, the attacker would not be able to map the debug registers into the virtual memory even when she has gained `root` privilege by tools like SuperSU [78]. Moreover, to prevent the attacker from replacing the stock kernel with a customized kernel that enables the LKM, the OEM may add the kernel into the secure boot chain and apply additional verification to 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.

6.3 A Practical Defense of NAILGUN Attack

As mentioned in Section 6.2.2, applying access control to the debug registers would be a possible solution to mitigate NAILGUN attack. In the ideal case, the restriction could be achieved by hardware to avoid any software manipulation in future devices. However, for the deployed devices, a software-based restriction would be more practical. Thus, we focus on designing a software-based mitigation mechanism in this section.

6.3.1 Threat Model

We assume the attacker owns non-secure EL1 privilege and aims to achieve privilege escalation via NAILGUN attack. Since the status and management mechanism of debug authentication signals varies in different SoCs, we make no assumption on the current status of the signals and consider the management mechanism is not available.

We assume that the Virtualization Extension is presented on the target platform. This extension is an optional feature for ARMv7-A architecture but becomes mandatory in ARMv8-A architecture. Moreover, we assume the software stack of the platform enforces a secure boot mechanism to avoid direct manipulation of the privileged software image.

6.3.2 Design

We aim to apply a software-based access control mechanism to avoid unexpected access to debug registers, and the privilege level to implement the mechanism is important to draw a balance between security and usability. If it is implemented in the non-secure EL1, malware with kernel privilege can bypass it easily since they own the same privilege. Once the defense is implemented in the secure state, it might introduce a significant performance overhead due to the semantic gap between the secure and non-secure state. In contrast, we find non-secure EL2 to be a good workaround. On the one hand, the high privilege prevents the malware from tampering with the defense mechanism; on the other hand, EL1&0 Stage 2 translation introduced in Section 2.6 would provide an efficient approach to restrict the access from EL1 to the debug register even without the semantics of EL1.

Fig. 10 shows the designed defense mechanism. The non-secure EL1&0 Stage 1 translation is typically used by the kernel to manage the virtual memory of applications, and we leave it unmodified. Instead, the Stage 2 translation is enabled as a gatekeeper to restrict the access to debug registers. When the Stage 2 translation is enabled, the output address of Stage 1 translation is considered as an IPA and not processed by the processor directly. The IPA is further translated to the PA with the Stage 2 translation table. Therefore, we leverage the Stage 2 translation table to mark the pages related to debug registers as invalid, which leads to a translation fault if these pages are accessed. With this design, even when an attack owns the EL1 privilege to manipulate the Stage 1 translation to access the debug registers, the Stage 2 translation stops the attack by the translation fault. Although an additional translation layer is enabled, we consider it is relatively lightweight compared to the traditional hypervisors (e.g., Xen [79], [80] and KVM [81]) since the workload in EL2 is minimized. Our evaluation in Section 6.3.4 also shows that the performance overhead introduced by the additional translation layer is negligible. Note that our defense mechanism can be directly transplanted to other ARM architectures with Virtualization Extension (i.e., ARM-A and ARMv8-R). For the ARM architectures that work with MPU, we can restrict the low-privilege access (PL0 in ARMv7-R, unprivileged mode in ARMv7-M, and non-secure modes in ARMv8-M) to debug registers with MPU configuration.

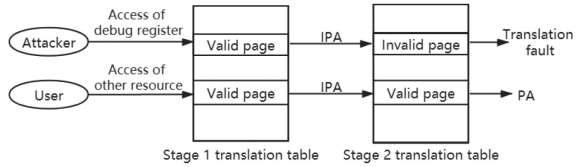


Fig. 10. Preventing NAILGUN via Stage 2 translation.

Moreover, we consider the proposed restriction would not affect regular applications and analysis systems mentioned in Challenge 1 of Section 6.1 since these applications and systems normally achieve debugging via the traditional debugging model with JTAG connections. In this model, the access to the debug registers relies on the DAP and does not go through the memory translation enforced by MMU. In the ARMv8.4-A architecture, ARM also suggests restricting the non-secure access to the debug registers performed in the inter-processor debugging model.

6.3.3 Implementation

To verify the defense mechanism, we implement a prototype on top of the official firmware released by Raspberry PI Foundation [82] and deploy it to a Raspberry PI 3 Model B+ board. Specifically, we enforce the defense mechanism before the booting of the official Linux kernel.

First, we reserve a continuous physical memory region at the bottom of the non-secure memory (`0x3effb000-0x3effffff`) for the Stage 2 translation table. To prevent the EL0&1 access to the debug registers and the created translation table, we clear bit[0] of the related table entries to mark the corresponding mapping as invalid. For other physical addresses, we use flat mapping to avoid additional engineering effort. Note that the physical addresses of the debug registers are decided by the SoC manufacturers, and the related addresses on the Raspberry PI can be found in the publicly available manual [83].

```
root@raspberrypi:~# insmod nailgun.ko
[   75.705543] nailgun: loading out-of-tree module taints kernel.
[   75.715408] 1. Unlock debug and cross trigger registers
[   75.722676] Unable to handle kernel paging request at virtual address 0badd9c1
[   75.731440] pgd = 580c61d9                    Translation fault when access sensitive registers
[   75.735519] [0badd9c1] *pgd=00000000
```

Fig. 11. Effectiveness evaluation of the defense mechanism.

Next, to enable the Stage 2 translation with the predefined translation table, we write the base address of the table (`0x3effb000`) to the `BADDR` bits of the Virtualization Translation Table Base Register (`VTTBR`) and set the `VM` bit of Hyp Configuration Register (`HCR`).

To reduce the lookup times of the translation table, we give priority to using 1GB and 2MB blocks in the translation table, and minimize the usage of 4k blocks. Note that although the 64-bit architecture introduces a larger memory address space, the minimal table lookup times in 32-bit architecture and 64-bit architecture are the same. Specifically, the table lookup times for 1GB, 2MB, and 4KB blocks in the Stage 2 translation are 1, 2, and 3 times, respectively. Therefore, we consider the performance overhead introduced by Stage-2 translation is similar in 32-bit and 64-bit architectures. However, the large address space in 64-bit architecture requires more reserved memory for the Stage-2 translation table since the number of entries increases. This might be mitigated by using coarse-grained Stage-2 translation tables (e.g., 64KB translation granule) in devices with limited memory.

6.3.4 Evaluation

Effectiveness Evaluation. To demonstrate the effectiveness of the defense, we launch NAILGUN attack mentioned in Section 5.2.2 on a Raspberry PI 3 Mobel B+ board equipped with the defense mechanism. As shown in Fig. 11, the access to the debug registers is terminated with an error message indicating a translation fault. It illustrates that the proposed defense mechanism has successfully prevented NAILGUN on the platform.

Performance Evaluation. Based on the implementation on Raspberry PI, we evaluate the performance of the proposed defense mechanism with three open-source benchmarks (i.e., *Nbench,nbench:zosxang*, *Sysbench* [85], and *Unixbench* [86]). In the evaluation, we run each benchmark 30 times with and without the defense mechanism, respectively. Performance scores without the defense mechanism are normalized as 1, while scores of the defense-enabled system are presented as a ratio to 1. The result shows that the performance overhead of our defense mechanism is less than 1.1% in all cases.

Unixbench [86] evaluates the performance of Unix-like systems with various tests (e.g., *I/O intensive* and *pipe throughput*) and provides a weighted performance score of the system. To achieve a comprehensive view, we leverage different options of *Unixbench* (i.e., one copy, four copies, and eight copies) to evaluate the performance of the system, and the result is shown in Fig. 12a. Specifically, the performance overhead of our defense mechanism is 0.30%, 0.54%, and 1.03% while running with one copy, four copies, and eight copies, respectively.

Nbench [84] evaluates the CPU performance with more than ten test cases. Fig. 12b shows the result of six cases, and the result of other tests are similar (less than 0.10% performance decrease). As shown in the figure, the performance drops by 0.41% in *Neural Net* test and less than 0.10% in other tests.

Sysbench [85] is a multithreaded benchmark with arbitrarily complex workloads. With *Sysbench*, we evaluate the performance of *Memory Reading*, *Memory Writing*, and *Prime Calculation* in our system. For *Prime Calculation*, we set the maximum number of the request as 20000. For *Memory Reading* and *Memory Writing*, we configure the total block size as 4KB, with the total memory size as 20GB. All tests run with 16 threads while the remaining settings are default. As indicated in Fig. 12c, the performance of *Memory Reading* and *Memory Writing* declines around 0.30%, and that of *Prime Calculation* drops about 0.01%.

7 CONCLUSION

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 investigate a series of ARM-based platforms powered by different SoCs and deployed in various product domains. Our investigation exposes 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-world devices with different ARM-A architectures show that almost all the ARM-A platforms we investigated are vulnerable to the attack. The analysis on ARM-R and ARM-M architecture shows that these architectures may also suffer from NAILGUN attack. Additionally, we discuss potential countermeasures to our attack from different layers of the ARM ecosystem to improve the security of the commercial devices. Finally, we design a lightweight defense mechanism based on ARM virtualization technology to restrict the access to the debug registers with a negligible performance penalty.

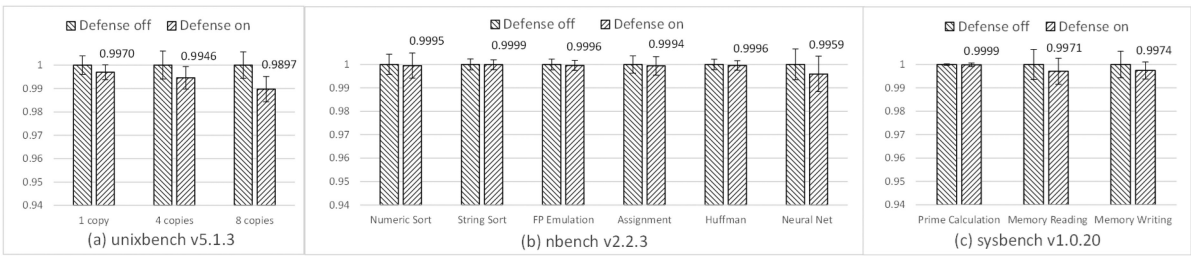


Fig. 12. Performance evaluation with different benchmarks.

REFERENCES

[1] Intel, "64 and IA-32 architectures software developer's manual," 2016. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[2] Intel, "Architecture instruction set extensions and future features programming reference," 2016. [Online]. Available: https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf

[3] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 167–182.

[4] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "Postmortem program analysis with hardware-enhanced post-crash artifacts," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 17–32.

[5] ARM, "Architecture reference manual ARMv7-A and ARMv7-R edition," 2011. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html

[6] ARM, "ARMv8-A reference manual," 2016. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.k/index.html

[7] ARM, "Embedded trace macrocell architecture specification," 2011. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/index.html

[8] ARM, "Embedded cross trigger," 2009, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0314h/Babhjchd.html

[9] IEEE, "Standard for test access port and boundary-scan architecture," 2013. [Online]. Available: https://standards.ieee.org/findstds/standard/1149.1–2013.html

[10] Intel, "System debugger," 2021. [Online]. Available: https://software.intel.com/en-us/system-studio/system-debugger

[11] ARM, "DS-5 development studio," 2010. [Online]. Available: https://developer.arm.com/products/software-development-tools/ds-5-development-studio

[12] OpenOCD, "Open on-chip debugger," 2009. [Online]. Available: http://openocd.org/

[13] ARM, "TrustZone security," 2009. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html

[14] Raspberry PI Foundation, "Model B+," 2018. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/

[15] miniNodes, "ARM servers," 2013. [Online]. Available: https://www.mininodes.com/

[16] Packet, "Cloud service," 2016. [Online]. Available: https://www.packet.net/

[17] Scaleway, "Cloud service," 2015. [Online]. Available: https://www.scaleway.com/

[18] Motorola, "Nexus 6," 2014. [Online]. Available: https://support.motorola.com/products/cell-phones/android-series/nexus-6

[19] Samsung, "Exynos processors," 2011. [Online]. Available: https://www.samsung.com/semiconductor/minisite/exynos/

[20] Hisilicon, "Kirin processors," 2012. [Online]. Available: http://www.hisilicon.com/en/Products

[21] Xiaomi, "Redmi 6," 2018. [Online]. Available: https://www.mi.com/global/redmi-6/

[22] Z. Ning and F. Zhang, "Understanding the security of ARM debugging features," in *Proc. 40th IEEE Symp. Secur. Privacy*, 2019, pp. 602–619.

[23] Zhenyu Ning, "Nailgun: Break the privilege isolation in ARM devices," 2019, [Online]. Available: https://github.com/ningzhenyu/nailgun

[24] ARM, "CoreSight components technical reference manual," 2009. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0314h/index.html

[25] Project Zero, "Lifting the (hyper) visor: Bypassing Samsung's real-time kernel protection," 2017. [Online]. Available: https://googleprojectzero.blogspot.com/2017/02/lifting-hyper-visor-bypassing-samsungs.html

[26] ARM, "Juno ARM development platform SoC technical reference manual," 2016. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0515b/

[27] 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 *Proc. 36th IEEE Symp. Secur. Privacy*, 2015, pp. 591–604.

[28] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proc. 11th ACM SIGSAC Symp. Inf. Comput. Commun. Secur.*, 2016, pp. 353–364.

[29] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 897–912.

[30] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 549–564.

[31] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. 36th IEEE Symp. Secur. Privacy*, 2015, pp. 605–622.

[32] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 1057–1074.

[33] NXP, "i.MX53 quick start board," 2011. [Online]. Available: https://www.nxp.com/docs/en/user-guide/IMX53QKSRTRQSG.pdf

[34] D. Zhang, "A set of code running on i.MX53 quick start board," 2014. [Online]. Available: https://github.com/finallyjustice/imx53qsb-code

[35] OpenSSL Software Foundation, "OpenSSL cryptography and SSL/TLS toolkit," 2017. [Online]. Available: https://www.openssl.org/

[36] ARM, "Trusted firmware," 2015. [Online]. Available: https://github.com/ARM-software/arm-trusted-firmware

[37] Linaro, "ARM development platform software," 2015. [Online]. Available: https://releases.linaro.org/members/arm/platforms/15.09/

[38] Huawei, "Ascend mate 7," 2014. [Online]. Available: https://consumer.huawei.com/en/support/phones/mate7/

[39] Motorola, "E4 plus," 2017. [Online]. Available: https://www.motorola.com/us/products/moto-e-plus-gen-4

[40] AmishTech, "Motorola E4 plus - More than just a fingerprint reader," 2017. [Online]. Available: https://community.sprint.com/t5/Android-Influence/Motorola-E4-Plus-More-Than-Just-a-Fingerprint-Reader/ba-p/979521

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

[42] R. Sasmal, "Fingerprint scanner: The ultimate security system," 2014. [Online]. Available: https://in.c.mi.com/thread-239612–1-0.html

[43] Fingerprints, "FPC1020 touch sensor," 2014. [Online]. Available: https://www.fingerprints.com/technology/hardware/sensors/fpc1020/

[44] Fingerprints, "Product specification FPC1020," 2014. [Online]. Available: http://www.shenzhen2u.com/doc/Module/Fingerprint/710-FPC1020_PB3_Product-Specification.pdf

[45] Z. Wu, "FPC1020 driver," 2015. [Online]. Available: https://android.googlesource.com/kernel/msm/+/9f4561e8173cbc2d5a5cc0fcda3c0becf5ca9c74

[46] Arm, "Fixed Virtual Platforms," 2014. [Online]. Available: https://developer.arm.com/tools-and-software/simulation-models/fixed-virtual-platforms

[47] Arm, "Arm architecture reference manual supplement Armv9, for Armv9-A architecture profile," 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0608/latest

[48] ARM, "ARM architecture reference manual supplement - ARMv8, for the ARMv8-R AArch32 architecture profile," 2020. [Online]. Available: https://developer.arm.com/documentation/ddi0568/latest/

[49] ARM, "Arm architecture reference manual supplement - Armv8, for Armv8-R AArch64 architecture profile," 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0600/latest/

[50] ARM, "ARMv7-M architecture reference manual," 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0403/latest/

[51] ARM, "ARMv8-M architecture reference manual," 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0553/latest/

[52] Dialog Semiconductor, "SmartBond DA1469x product family," 2019. [Online]. Available: https://www.dialog-semiconductor.com/products/da1469x-product-family

[53] NXP, "LPC552x/S2x: Mainstream Arm Cortex-M33-based microcontroller family," 2019. [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc5500-cortex-m33/lpc552x-s2x-mainstream-arm-cortex-m33-based-microcontroller-family:LPC552x-S2x

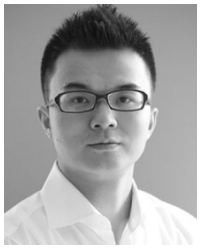
[54] Nordic Semiconductor, "nRF9160," 2020. [Online]. Available: https://infocenter.nordicsemi.com/pdf/nRF9160_PS_v2.0.pdf

[55] Renesas Electronics Corporation, "48MHz Arm Cortex-M23 ultra-low power general purpose microcontroller," 2020. [Online]. Available: https://www.renesas.com/us/en/products/microcontrollers-microprocessors/ra-cortex-m-mcus/ra2l1–48mhz-arm-cortex-m23-ultra-low-power-general-purpose-microcontroller
[56] STMicroelectronics, "STM32L5 Series," 2020. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32l5-series.html
[57] Microchip Technology Inc., "World-Class, Award-Winning SAM L10 and SAM L11 Microcontroller Family," 2018. [Online]. Available: https://www.microchip.com/design-centers/32-bit/sam-32-bit-mcus/sam-l-mcus/sam-l10-and-l11-microcontroller-family
[58] NXP, "LPC55S69-EVK: LPCXpresso55S69 Development Board," 2019. [Online]. Available: https://www.nxp.com/design/development-boards/lpcxpresso-boards/lpcxpresso55s69-development-board:LPC55S69-EVK
[59] Nordic Semiconductor, "nRF5340," 2020. [Online]. Available: https://infocenter.nordicsemi.com/pdf/nRF5340_PS_v1.0.pdf
[60] D. Balzarotti *et al.*, "An experience in testing the security of real-world electronic voting systems," *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, pp. 453–473, Jul./Aug. 2010.
[61] 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 *Proc. 20th USENIX Secur. Symp.*, 2011, Art. no. 4.
[62] L. Cojocar, K. Razavi, and H. Bos, "Off-the-shelf embedded devices as platforms for security research," in *Proc. 10th Eur. Workshop Syst. Secur.*, 2017, Art. no. 1.
[63] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 309–326.
[64] 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 *Proc. 24th Netw. Distrib. Syst. Secur. Symp.*, 2017, Art. no. 55.
[65] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling near-real-time dynamic analyses of embedded systems," in *Proc. 9th USENIX Workshop Offensive Technol.*, 2015, Art. no. 7.
[66] 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 *Proc. 4th Workshop Hardware Architect. Support Secur. Privacy*, 2015, Art. no. 3.
[67] S. Mazloom, M. Rezaeirad, A. Hunter, and D. McCoy, "A security analysis of an in-vehicle infotainment and app platform," in *Proc. 10th USENIX Workshop Offensive Technol.*, 2016, pp. 232–243.
[68] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on ARM," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 33–49.
[69] J. Zaddach *et al.*, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *Proc. 21st Netw. Distrib. Syst. Secur. Symp.*, 2014, Art. no. 8.
[70] A. Abbasi, T. Holz, E. Zambon, and S. Etalle, "ECFI: Asynchronous control flow integrity for programmable logic controllers," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, 2017, pp. 437–448.
[71] Z. B. Aweke *et al.*, "ANVIL: Software-based protection against next-generation rowhammer attacks," in *Proc. 21st ACM Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2016, pp. 743–755.
[72] J. Demme *et al.*, "On the feasibility of online malware detection with performance counters," in *Proc. 40th ACM/IEEE Int. Symp. Comput. Archit.*, 2013, pp. 559–570.
[73] 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 *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 1075–1091.
[74] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "SoK: The challenges, pitfalls, and perils of using hardware performance counters for security," in *Proc. 40th IEEE Symp. Secur. Privacy*, 2019, pp. 20–38.
[75] F. Zhang, K. Leach, A. Stavrou, and H. Wang, "Using hardware features for increased debugging transparency," in *Proc. 36th IEEE Symp. Secur. Privacy*, 2015, pp. 55–69.
[76] S. Skorobogatov, "Fault attacks on secure chips: From glitch to flash," 2011. [Online]. Available: https://www.cl.cam.ac.uk/sps32/ECRYPT2011_1.pdf
[77] NXP, "i.MX53 multimedia applications processor reference manual," 2012. [Online]. Available: https://cache.freescale.com/files/32bit/doc/ref_manual/iMX53RM.pdf
[78] J. Jongma, "SuperSU," 2012. [Online]. Available: https://android.googlesource.com/kernel/msm/+/9f4561e8173cbc2d5a5cc0fcda3c0becf5ca9c74.
[79] Xen project, "Xen ARM with virtualization extensions," 2014. [Online]. Available: https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions.
[80] J.-Y. Hwang *et al.*, "Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones," in *Proc. 5th IEEE Consum. Commun. Netw. Conf.*, 2008, pp. 257–261.
[81] C. Dall and J. Nieh, "KVM/ARM: The design and implementation of the Linux ARM hypervisor," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2014, pp. 333–348.
[82] Raspberry PI Foundation, "Raspbian kernel source code," 2021. [Online]. Available: https://github.com/raspberrypi/linux
[83] Raspberry PI Foundation, "Raspberry PI Processors," 2021. [Online]. Available: https://www.raspberrypi.com/documentation/computers/processors.html
[84] U. F. Mayer, "Nbench," 2014. [Online]. Available: https://github.com/zosxang/nbench
[85] Alexey Kopytov, "Sysbench," 2014. [Online]. Available: https://github.com/akopytov/sysbench
[86] Byte Magazine, "Unixbench," 2015. [Online]. Available: https://github.com/kdlucas/byte-unixbench

Zhenyu Ning received the PhD degree in computer science from Wayne State University, Detroit, Michigan, in 2020. He is a research assistant professor with the Department of Computer Science and Engineering, Southern University of Science and Technology (SUSTech). His research interests include security and privacy, including system security, mobile security, IoT security, trusted execution environment, hardware-assisted security.

Chenxu Wang received the bachelor's degree in computer science and engineering from the Southern University of Science and Technology (SUSTech), Shenzhen, China. He is currently working toward the Joint PhD degree in computing at the Hong Kong Polytechnic University, Hong Kong. His research interests include virtualization and trusted execution environment on arm architecture.

Yinhua Chen received the bachelor's degree in computer science and engineering from the Southern University of Science and Technology (SUSTech), Shenzhen, China. He is currently working toward the master's degree in computer science and engineering at the Southern University of Science and Technology, Shenzhen, China. His research interests include system security and virtualization.



Fengwei Zhang is currently an associate professor with the Department of Computer Science and Engineering, Southern University of Science and Technology (SUSTech). His primary research interests include the areas of systems security, with a focus on trustworthy execution, hardware-assisted security, debugging transparency, and plausible deniability encryption. Before joining SUSTech, he spent four years as an assistant professor with the Department of Computer Science, Wayne State University.

Jiannong Cao is currently the Otto Poon Charitable Foundation professor in data science and the chair professor of distributed and mobile computing with the Department of Computing, Hong Kong Polytechnic University (PolyU). His research interests include distributed systems and blockchain, wireless sensing and networking, big data and machine learning, and mobile cloud and edge computing.