CACHE-BASED ATTACK AND DEFENSE ON ARM PLATFORM

by

NAIWEI LIU, M.Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

COMMITTEE MEMBERS:
Ravi Sandhu, Ph.D., Chair
Meng Yu, Ph.D.
Ali Tosun, Ph.D.
Jianhua Ruan, Ph.D.
Mauricio Gomez, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
December 2020

DEDICATION

This dissertation is dedicated to my family members, for their love and support, especially in this special year.

ACKNOWLEDGEMENTS

First of all, I want to express my sincere gratitude toward my advisors, Dr. Ravi Sandhu and Dr. Meng Yu, for patient advising and dedicated teaching. They are hard-working researchers and carrying out high-level research with passion and efficiency. I learned a lot from there working ethics and lifestyle. I am always in debt to these valuable teaching and supervising.

I also want to thank Dr. Jianhua Ruan, Dr. Mauricio Gomez and Dr. Ali Tosun for taking time to be my committee members. I am honored to have you all.

This dissertation and research project on ARM security is partially supported by NSF CNS-1634441. Some of the contents are published in EAI Journal on Security and Privacy in 2019, and WISA conference 2020.

This Masters Thesis/Recital Document or Doctoral Dissertation was produced in accordance with guidelines which permit the inclusion as part of the Masters Thesis/Recital Document or Doctoral Dissertation the text of an original paper, or papers, submitted for publication. The Masters Thesis/Recital Document or Doctoral Dissertation must still conform to all other requirements explained in the Guide for the Preparation of a Masters Thesis/Recital Document or Doctoral Dissertation at The University of Texas at San Antonio. It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this Masters Thesis/Recital Document or Doctoral Dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts, which provide logical bridges between different manuscripts, are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the students contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the Masters Thesis/Recital Document or Doctoral Dissertation attest to the accuracy of this statement.

December 2020

CACHE-BASED ATTACK AND DEFENSE ON ARM PLATFORM

Naiwei Liu, Ph.D.
The University of Texas at San Antonio, 2020

Supervising Professors: Ravi Sandhu, Ph.D. and Meng Yu, Ph.D.

In recent years, research efforts had been made to investigate a safe and secure environment for ARM platform. The new ARMv8 design had brought in with the design features that allow the secure design. However, there are still some security problems with both Cortex-A and Cortex-M series on ARMv8. For example, on Cortex-A series, there are risks that the system gets down on side-channel attacks. One major category of side-channel attacks utilizes cache memory to obtain a victim's secret information. In the cache based side-channel attacks, an attacker measures a sequence of cache operations to obtain a victim's memory access information, deriving more sensitive information. The success of such attacks highly depends on accurate information about the victim's cache accesses. Cortex-M series, on the other hand, have some design so that the side-channel attack can be prevented, but it also needs a security wrapper to ensure the security of the users' privacy data. In this thesis, we discuss the defense against side-channel attack on Cortex-A series chips, and also the security design on Cortex-M series chips. Our adaptive noise injection can significantly reduce the bandwidth of side-channel while maintaining affordable system overheads. The proposed defense mechanisms can be used on ARM Cortex-A architectures, and also in virtualized environments. Our experimental evaluation and theoretical analysis show the effectiveness and efficiency of our proposed defense.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: INTRODUCTION

1.1 Introduction

In Recent years, many research papers have been focusing on security design on ARM platform. Some of security framework are designed and implemented making use of TrustZone, a secure enclave provided by ARM on both Cortex-A and Cortex-M series. These defense frameworks target to memory protection, process protection and even cache protection. For example, some of the malicious applications or users can utilize the entry/exit of the TrustZone on ARM Cortex-A, launching a cache-based attack, and compromising the message channel between users and the system. As a result, some research papers target to this problem using access control of entry/exit operations, and some papers use isolated cache protection design. The research papers and their implementations can cut down the bandwidth of cache-based attack, with various level of overhead on the whole system.

On the other hand, there are more types of threats nowadays on IoT devices, mobile devices and even all the 'smart devices' based on ARM chips. On the first step, some research studied the last-level cache(LLC) threats on both single device and cloud with multiple devices [40] [17] [43] [12]. These attacks are very effective to extract users' private information without administrator's privileges. When setting up side channel attacks, an attacker collects the information of the victim's performance, power consumption, timing, etc. The collected information can be used to further derive more information about the victim, e.g., cryptographic keys, data being accessed, and so on. For example, memory access time can be very different depending on if the accessed data is in the cache. Thus, the data being accessed by the victim can be partially derived based on the data access time if the attacker and the victim are sharing data in the cache.

In a cloud computing system, the LLC is shared among multiple processor cores, making it vulnerable to LLC based side-channel attacks. Unlike L1 cache, LLC is much slower than L1 cache, leading to more difficult set up for side channels. There are different ways to launch side-channel attacks, e.g., FLUSH+RELOAD [40], [12], PRIME+PROBE [12, 16, 20], and bus-locking [37].

As an example, the FLUSH+RELOAD involves three steps. The attacker first flushes one or more of the desired cache contents using processor-specific instructions (e.g. `clflush` on x86 processors). Second, the attacker waits for sufficient time for the victim to use (or not to use) the flushed cache area. Finally, the attacker reloads previously flushed cache lines, measuring the reload time for each one of them to infer if it was touched by the victim. FLUSH+RELOAD strategy has been proven very effectively in many side channel attacks on x86 architecture. For example, Gülmezoğlu et al. [12] recovered the AES key of OpenSSL within 15 seconds.

Defense mechanisms using hardware designs [4, 19, 23, 35, 36] or software modifications [5, 8, 9, 20, 32] have been developed to mitigate the LLC based side channel attacks. Though very powerful, the hardware solutions require special features that are not available on commodity computer systems. Software solutions include software diversity transform [9], adding noise into the application [22, 32, 44], isolation through better scheduling [34], and others. However, most of the solutions are application specific and incur substantial performance overhead.

Moreover, LLC based side-channel attacks and defenses are mainly implemented and evaluated on the x86 architecture. While more and more mobile devices, smart phones, and Internet of Things (IoT) devices mainly use ARM architecture rather than the x86. Whether the side-channel attacks and defense mechanism are the same on the ARM based devices are not fully investigated.

On the ARM architecture, it is very different to construct side-channel attacks and defense mechanisms. For example, on an ARM platform, a cache flush operation is a privileged operation. It is a more secure design than x86 since a regular user has no access to cache FLUSH operations. Furthermore, when flush instructions are requested by a user, the system can invalidate the cache contents and FLUSH all the TLB entries, making LLC based side-channel attacks impossible, while considerable performance loss is introduced at the same time.

In this dissertation, we investigate the defense effectiveness to LLC based side-channel attacks on the ARM architecture and also propose a defense mechanisms for both ARM Cortex-A and ARMv8-M architectures. We use adaptive FLUSH operations on exit operations from TrustZone. We carefully add cache operations to the system such that the measurement of the victim's mem-

ory access time becomes very difficult or even impossible. As a result, the bandwidth of the side channel is significantly reduced, making the attacker unable to compromise the device with an acceptable cost. We implement and evaluate the proposed defense on both ARMv8-M and Cortex-A series chips. The experimental results show that our proposed defense is effective in both mitigating the cache-based side-channel attacks and supporting efficient execution of normal applications.

The design and implementation of defense have to overcome several challenges. First, on ARM architecture, there are different banked registers and modes. FLUSH+RELOAD operations are designed as privileged operations but they are supported by virtualization software in different ways. We will look at both the threat and defense in such context. Second, we target at protection of the whole system instead of a specific application while providing affordable overheads. Most existing software solutions are either specific to an application or have substantial overhead. Third, it is challenging to adaptively inject FLUSH operations with affordable overhead to the system and normal applications.

In summary, this thesis has the following contributions:

- We investigate the cache-based side-channel attacks on ARM architecture.
- We design and implement a defense mechanism against several types of side-channel attacks on ARM platforms. The proposed defense is adaptive, effective, and efficient. The protection can work for a whole system rather than a specific application.
- We have done experimental evaluation for our protection mechanisms. The evaluation results show effectiveness and efficiency of our design.
- Our protection can be implemented in either a tool in the system, or a built-in part in an application.

Chapter 2: LITERATURE REVIEW

2.1 Cache-Based Attack

In a cloud computing system or a computer with multiple processes and threads, the Last Level Cache (LLC) is shared among multiple processor cores, making it vulnerable to LLC based side-channel attacks. Unlike L1 cache, LLC is much slower than L1 cache, leading to more difficult set up for side channels. There are different ways to launch side-channel attacks, e.g., FLUSH+RELOAD [40], [12], PRIME+PROBE [12] [16] [20], and bus-locking [38].

For example, the FLUSH+RELOAD involves three steps. The attacker first flushes one or more of the desired cache contents using processor-specific instructions (e.g. clflush on x86 processors). Second, the attacker waits for sufficient time for the victim to use (or not to use) the flushed cache area. Finally, the attacker reloads previously flushed cache lines, measuring the reload time for each one of them to infer if it was touched by the victim. FLUSH+RELOAD strategy has been proven very effectively in many side channel attacks on x86 architecture. For example, Gulmezoglu et al. [12] recovered the AES key of OpenSSL within 15 seconds. Yarom and Falkner [40] recover a RSA encryption key across VMware VMs using FLUSH+RELOAD attack, and Irazoqui et al. [17] recovered AES keys using similar attack and exploiting the vulnerabilities in cache. For PRIME+PROBE attack, Work [20] recover AES keys in a cross-VM Xen 4.1 using PRIME+PROBE attack. Liu et al. [19] presented a PRIME+PROBE type side-channel attack model against the LLC, which is tested to be practical and threatens the system.

2.2 Cache-based Covert Channels

A covert channel can be created through sharing resources. The higher bandwidth is in the cover channel, the faster the information leakage can achieve. Ristenpart et al. [26] experimented with L2 covert channels in a cloud environment. Their bandwidth is around 0.2 bps. Xu et al. [37] extended this attack. The capacity of L2 cover channel is 233 bps. Percival demonstrated that shared access

to memory caches provides a high bandwidth covert channel between threads in [24]. The capacity of L2 covert channel is approximately 100 kbps. Wu et al. [38] presented a new covert channel attack with high-bandwidth (over 190.4 kbps) and reliable data transmission in the cloud. Liu et al. presented a PRIME+PROBE side-channel attack, achieving a bandwidth of 1.2 mbps [20].

By accurately mapping the cache sets, our attack achieves a much higher bandwidth than prior work.

2.3 Last-level Cache(LLC) Side-Channel Attacks

Due to the low channel capacity, an LLC-based side-channel typically only leaks course-grain information. For example, the attacks of Ristenpart et al. [26] leak information about co-residency, traffic rates and keystroke timing. Zhang et al. [42] use an L2 side-channel to detect non-cooperating co-resident VMs. Our attack improves on this work by achieving a high granularity that enables leaking of cryptographic keys. Yarom and Falkner (FLUSH+RELOAD) [40] show that when attacker and victim share memory, e.g. shared libraries, the technique of Gullasch et al. [11] can achieve an efficient crossVM, cross-core, LLC attack. Side-channel attack removes the requirement for sharing memory, and is powerful enough to recover the key from the latest GnuPG crypto software which uses the more advanced 618 sliding window technique for modular exponentiation, which is impossible using FLUSH+RELOAD attacks. In concurrent work Irazoqui et al. [16] describe the use of large pages for mounting a synchronous LLC PRIME+PROBE attack against the last round of AES.

Recently, many research work on side-channel attacks in a Trusted Execution Environment (TEE), such as Intel SGX and ARM Trustzone [21, 28]. There are some other types of side-channel attacks based on different shared data or data structures in the system. For example, Xu et al. [39] introduced controlled-channel attacks, a new type of side-channel attack. The attack allows an untrusted operating system to extract large amounts of sensitive information from protected applications on systems, such as Overshadow [7], InkTag [13], or Haven [3]. This attack is not based on LLC, but based on the page accessed by the VMs. Our techniques do not apply directly

to these attacks but the idea of noise injection can still be used theoretically.

Basically, the difference between a covert channel and a side-channel is the role of the attacker side. In a covert channel, the attacker trying to get the encrypted message can be either side of the channel, possibly the sender or the receiver. However, in a side-channel, the attacker is on a third side, trying to listen to the message channel to steal information. Both the sender and the receiver can be unaware of the existence of the malicious user.

2.4 Manipulating Cache Contents

Two types of LLC-based side-channels have been extensively studied recently. One is the FLUSH+RELOAD [12, 17,40,43], and the other is PRIME+PROBE [12,16,20]. In FLUSH+RELOAD, the attacker and victim share a physical memory page, such as sharing libraries. In [42], the adversary was able to conduct a cache-based attack to track the execution path of a victim and extract a secret of interest from the victim. Yarom and Falkner [40] applied the attack to recover a RSA encryption key across VMware VMs, and Irazoqui et al. [17] recovered AES keys. PRIME+PROBE can be conducted when the attacker and victim share the same CPU cache sets. Liu et al. presented an effective and practical implementation of the PRIME+PROBE side-channel attack against the last-level cache in [20] . Work [16] implemented PRIME+PROBE to recover AES keys in a cross-VM setting on Xen 4.1.

It is proven that the FLUSH+RELOAD technique is particularly effective when memory duplication features are enabled by the VMM [12, 17]. Gülmezoğlu et al. applied FLUSH+RELOAD attack on OpenSSL implementation of AES, and recovered the key in just 15 seconds working across cores in a cross-VM setting [12]. In this thesis, we mainly focus on FLUSH+RELOAD technique and our proposed techniques can also be applied to PRIME+PROBE using the same principle.

The FLUSH and RELOAD technique is a variant of PRIME+PROBE that relies on sharing pages between the attacker and the victim processes. With shared pages, the malicious user can ensure that a specific memory line is evicted from the whole cache hierarchy. The attacker uses this to monitor access to the memory line. The attack is a variation of the technique suggested by

Gullasch et al. [11], which include adaptations to multi-core and virtualized environments.

A round of attack consists of three phases. During the first phase, the monitored memory line is flushed from the cache hierarchy. The attacker, then, waits to allow the victim time to access the memory line before the third phase. In the third phase, the attacker reloads the memory line, measuring the time to load it. If during the wait phase the victim accesses the memory line, the line will be available in the cache and the reload operation will take a short time. If, on the other hand, the victim has not accessed the memory line, the line will need to be brought from memory and the reload will take significantly longer.

The victim access can overlap the reload phase of the attacker. In such a case, the victim access will not trigger a cache fill. Instead, the victim will use the cached data from the reload phase. Consequently, the attacker will miss the access.

2.5 Defense Mechanisms

Many defenses using hardware designs [4, 19, 23, 35, 36] or software modifications [5, 8, 9, 30, 32] have been developed to mitigate the LLC based side channel attacks. The hardware solutions though very powerful, require special hardware features. There are different software solutions, such as software diversity transform [9], adding noise into the application [22, 32, 44], and soft isolation through better scheduling [34].

2.6 Noise Injection based Defense

Page [22] suggested manually adding noise, such as garbage instructions, and random loads, into the encryption routine to make cache side-channel attacks more difficult. The proposed approach is specific to encryption application and incurs substantial performance overhead. Tromer et al. [32] suggested several countermeasures for the side channel attack, including injecting noise to the memory access pattern by adding spurious accesses, e.g., by performing a dummy encryption in parallel to the real one. This would decrease the signal visible to the attacker. However, they do not give any detailed design or implementation.

Zhang and Reiter [44] designed and implemented a defense system called Düppel that enables a tenant virtual machine to defend itself from cache-based side-channel attacks in public clouds. A tenant can automatically inject additional noise into the timings that an attacker might observe from caches. Since these timings are commonly used by an attacker to infer the sensitive information about a victim VM, injecting noise into them will generally make the attacks more difficult. The solution requires users to identify the particular processes that should be protected [45]. Our approach generally protects the system and does not need user to identify any specific process.

2.7 Other Types of Defense

Zhou et al. [45] proposed a memory copy approach to dynamically manage physical memory pages shared between security domains to disable sharing of LLC lines, preventing FLUSH+RELOAD side channels via LLCs. In their proposed work, a victim's access to its copy will be invisible to an attacker's RELOAD in a FLUSH+RELOAD attack. Varadarajan et al. [34] investigated a soft isolation, reducing the risk of sharing through better scheduling design. It is also possible to limit the frequency of potentially dangerous interactions between mutually untrustworthy programs [41].

Compared with the above work, our approach is easy to deploy and effective, and provides protection to an entire system rather than a specific application. Moreover, all the above defense systems are implemented on x86 platform. Our work is focusing on the LLC-based attack and defense on both ARM architecture.

2.8 Hardware-Based Defense

Bernstein [4] suggested to add L1-table-lookup instruction to load an entire table in L1 cache, and also load a selected table entry in a constant number of CPU cycles. Page [23] investigated a partitioned cache architecture. Wang and Lee [35] [36] [28] proposed new security-aware cache designs to thwart the LLC side channel attack with low overhead. In [36], the Partition-Locked cache (PLcache) was able to lock a sensitive cache partition into cache, and Random Permutation cache (RPcache) randomized the mapping from memory locations to cache sets. In [19], a

novel random fill cache architecture that replaces demand fetch with random cache fill within a configurable neighborhood window was proposed. While the hardware solutions provide strong isolations between the victim and the attacker, they require special hardware features that are not immediately available form commodity processors.

2.8.1 Software-Based Defense

Some researchers proposed to modify applications to better protect secrets from side-channel attacks. Brickell et al. [5] proposed three individual mitigation strategies: compact S-box table, frequently randomized tables, and pre-loading of relevant cache-lines. It compressed and randomized tables for AES. However, it requires manually rewriting the AES implementation and is specific to AES. Cleemput et al. [8] applied the mitigating code transformations to eliminate or minimize key-dependent execution time variations. Crane et al. [9] proposed a software diversity technique to transform each program unique. The approach offers probabilistic protection against both online and off-line side-channel attacks. In their work, using function or basic-block level dynamic control-flow diversity along with static cache noise results in a performance slowdown of 1.76x-2.02x compared to the baseline AES encryption when using 10%-50% cache noise insertion. Dynamic cache noise at 10%-50% has significantly impact on performance (2.39-2.87x slowdown). However, above software solutions are typically application specific or incur substantial performance overhead.

2.9 Recent Research on ARM-based Defenses

On the year 2017, Sandro Pinto and some other researchers proposed LTZVisor [25], which is based on TrustZone to protect and assist ARM virtualization. They implement and test on ARM platform and have an overhead of around 22% at the highest user switching frequency. Guan el al. proposed TrustShadow [10], using TrustZone to protect user's applications, with little or no change on the application itself. The overhead here is around 10% at worst case and 2% on average. However, the framework is not tested on ARMv8-M, which has different structure and instruction

sets from ARM Cortex-A series. Similar as LTZVisor, Hua et al. designed and implemented vTZ [15], a virtualization based defense framework on ARM. The overhead of vTZ is on average case around 5%.

According to their work, the most popular solution on ARM is virtualization, using TrustZone to protect the application, data and user's private keys. This can only be implemented on ARM Cortex-A series, which has different level of cache, multiple privilege levels and powerful CPU. On ARMv8-M series, however, similar implementation is not applicable. On ARMv8-M series chips, there is no cache on the structure, and normally the protection cannot be complicated due to the limited resource on the devices. Compared with their work, we have a more directly protection, with acceptable overhead and good performance.

2.10 Recent Research on ARM TrustZone

In recent years, some papers have discussions and new research findings on ARM platform, especially focusing on TrustZone protection. Zhang et al. [46] proposed an Android protection framework using TrustZone on ARM, protecting VoIP phone calls. It enclaves privacy data so the phone calls cannot be intercepted easily by malicious eavesdropping. Amacher et al. [1] have evaluate the performance of ARM TrustZone using TEEs and different benchmarks, but the security concern is out of that paper's scope. Keystone defense framework proposed by Dayeol Lee and others [**?**] is a good example of defense framework based on TrustZone. It enclaves protected operations and disables sharing in TLBs and memory blocks so there's no side-channel attack based on the vulnerability here. However, the timing side-channel attack is out of that paper's scope. In our discussion, there are still risks of side-channels when exiting from TrustZone, so we need also investigate the vulnerability at the gate of security enclave.

Chapter 3: OVERVIEW

3.1 ARM Overview

As multi-core processors become pervasive and the number of on-die cores increases, a key design issue facing processor architects is the security layers and policies for the on-die LLC. With LLC techniques, a CPU might only need to get around 5% data from main memory, which can improve the efficiency of CPU largely. In our implementations, we are using Intel i7-4790 processor, with 8Mb SmartCache. On ARMv8 Cortex-A platform, we are using Juno r1 Development Platform which has one A57 and one A53 processors on the board. A57 has a 2M LLC on the processor. On Cortex-M platform, we are using ARM Cortex-M4 series chips, the development platform has 3 pipeline stages and no built-in cache.

With the increasing complexity of computing systems, as well as multiple level of memory access, some registers are designed to store some specific hardware events. These registers are usually called hardware performance counters. We have many tools getting information from those performance counters, thus getting the performance information.

In our implementation, we use perf to collect the execution information of the programs. However, we cannot use perf for collecting timing information of memory access, since it cannot be accurate enough. On this thesis we use inline assemblies and consult some related registers to measure time associated information with our side-channels.

3.1.1 ARM Platform Introduction

Environment Overview

As multi-core processors become pervasive and the number of on-die cores increases, a key design issue facing processor architects is the hierarchy and policies for the on-die LLC. With LLC techniques, a CPU might only need to get around 5% data from main memory, which can improve the efficiency of CPU largely. On ARM, we are using Juno r1 Development Platform which has one

A57 and one A53 processors on the board. A57 has a 2M LLC on the processor.

With the increasing complexity of computing systems, as well as multiple level of memory access, some registers are designed to store some specific hardware events. These registers are usually called hardware performance counters. We have many tools getting information from those performance counters, thus getting the performance information.

In our implementation, we cannot use perf for collecting timing information of memory access on ARM, since it cannot be accurate enough, and not applicable on ARM. On this thesis we use inline assemblies to measure time associated information with our side-channels.

Process Structures

On implementations at ARM platform, the model contains with a sender, a receiver and an OS module to randomly inject cache flushes to generate noises into the channel. If the sender and the receiver are both from the attacker, it is a typical covert channel. If the sender program is a legitimate program, it is a typical side-channel configuration.

In this thesis, a channel is constructed and evaluated. The sender here sends a message in the stream. The receiver, on the other hand, analyzes the access time to the memory shared with the sender to figure out what is being sent. After receiver receiving the message, we study the quality of such channel in terms of bandwidth, accuracy with noises injections to the message channel.

We also have the error correction in the message channel. On this thesis, we use CRC for this purpose. The message passed through the message channel are checked using CRC, and when noises injected, the receiver uses CRC to try fixing the message, working to recover the message that the sender is sending.

Attack Based on ARM Platform

Attacks using shared resource based side-channels need to monitor the victim's activities on the shared resource. Using cache as an example, in a FLUSH+RELOAD attack, the attacker firstly FLUSHes specific cache lines, and waits for a predetermined time to RELOAD the contents. By

measuring the reload time, the attacker can learn if the shared contents with the victim have been used or not, thus deriving sensitive information about the victim.

Similarly, in a PRIME+PROBE attack, the attacker first measures the data reading time, and loads memory contents (PRIME) to a number of cache sets. The attacker then measures the access time to see if the data is accessed by the others (PROBE). The success of such side-channel attacks is highly depending on the following three necessary conditions: (1) the ability to precisely measure the memory access time; (2) the ability to selectively manipulate cache contents; and (3) sharing memory contents with the victim.

On both x86 and ARM architecture, there are performance counter registers and related machine instructions to obtain accurate time measurement to satisfy condition (1). Condition (3) can be easily satisfied since a modern operating system has a lot of shared memory pages through the shared libraries, code segments, etc. The challenge is to satisfy condition (2) on ARM architecture because the instructions to manipulate cache contents are privileged instructions that are not available to the regular users. If a user is at a privileged level, side-channel attacks are unnecessary. Thus, ARM architecture is secure by design if a single operating system is running on the processor. However, these support on ARM may open the door to side-channel attacks due to handling of cache operations.

Background on Different Structures of ARM

As mentioned above, on this thesis, we focus on ARM Cortex-A structure. However, devices and users using ARM Cortex-M structure are in a rising trend of numbers. On ARMv8-M, it does not have cache and memory mapping. Instead, it uses direct allocation on memory to ensure high performance. The memory on ARMv8-M is separated into different parts for different purposes.

As a result, the TrustZone entry and exit operations are with high efficiency, costing less than 10% of clock cycles comparing with Cortex-A series. On the other hand, the design of ARMv8-M made it difficult for design of defense. As devices using this structure usually with a simple or almost no OS, traditional defense framework are not applicable on those devices.

Based on our experiments and discussions, we can only focus on Cortex-A defense. For Cortex-M based defending framework, we focus on that topic on some other papers.

3.2 Design on ARM Cortex-A

According to our evaluation on current on-the-market systems and applications, we find out that more and more Trusted Execution Environment (TEE) technologies are being used on the implementations of secure system. Besides, most of the implementations are utilizing ARM TrustZone to protect the memory access and critical data. As we are interested in the performance overhead of defending using FLUSH operations on exiting TrustZone, the experiments should start from the measurements of using TrustZone, like the time cost and performance overhead.

Our experiments on ARM Cortex-A are in three different steps. For the first step, we test the cost of entering and exiting from TrustZone. After we get the exact data (clock cycles) related to TrustZone, the next step is to measure how much it takes up for the TEEs to call TrustZone related instructions or operations. On the third step, we try to clean the cache every time the system exiting from TrustZone, and see the performance overhead by these FLUSH operations added to the system. As the cache gets FLUSHed every time after the using of TrustZone, the risk of being side-channel attacked can be theoretically cut down to non-exist.

3.3 Overview on ARM Cortex-M

Unlike ARM Cortex-A series chips, M-series chips have different structure, and with other limitations. Most IoT devices are based on Cortex-A platform, but still a rising trend that more products are using Cortex-M platform. As a result, it is still valuable to investigate the defense against malicious attackers with TrustZone. In this thesis, we have similar tests on ARMv8-M platform, measuring the performance of TrustZone, as well as FLUSH operation overhead. Our experiments on Cortex-M are using ARM Versatile V2M-MPS2 Motherboard with ARM Cortex-M4 cores. It offers 8Mb of single cycle SRAM, and 16Mb of PSRAM. It supports the application of different ARM Cortex-M classes, from Cortex-M0, to M3, M4, and M7. Besides these support, the

development board supports simulation of ARMv8-M.

As mentioned above, on Cortex-M4 series chips, there is no built-in cache. However, the memory structure on M4 is different from other structures like x86 and Cortex-A. On that platform, memory blocks are allocated in fixed order, taking their assigned responsibilities. It is quite different from dynamic allocation, and is to the consideration of power consumption and performance overhead. Among these memory blocks, some are acting as 'cache-in-memory', so we can still see them working like cache and operate some instructions to read the working status of it.

The experiments are in two different steps. First, we measure the time cost entering and exiting from TrustZone. Next, we implement a program with TrustZone entry/exit instructions, as well as protected running steps. We then test it with controlling of the frequency of entry/exit instructions. We measure the FLUSH operation overhead according to different frequencies, and discuss the defense using FLUSH when exiting from TrustZone.

3.4 Threat Model and Assumptions

Side-channel attackers and other cache-based attackers are not based on compromised OS. They perform as 'man in the middle' and collecting time stamps of cache read/write operations. As discussed above, for side-channel attack, the processes do not need shared memory, so the model here has no assumption that they have to share memory in whole or in part. Because of the difference in the definition between side-channel and covert channel, on covert channel, it is possible that the attacker and victim share some resources, making a slight difference on the assumption. In our design of defense, we can efficiently decrease the bandwidth of both side-channel and covert channel, but we are testing the defense using side-channel attack model, so we assume that the memory is not shared between victim process and the attacker.

On system side, we assume that the operating system components in TrustZone is not compromised so that the attackers are forced to use covert channels or side channels without explicitly violating access control policies enforced by the operating system or other protection mechanisms. Besides that, we also assume the system is having a control part, i.e. handler to inject interference

into possible side-channel. Some instructions using assembly code is privileged to higher level to launch, so we assume they have the privilege level to inject noise. On the other hand, we assume that the noise injection process is not compromised, so the injection of noise is just for the defense, not for other malicious using like probing the cache.

We also assume that the attacker has sufficient privilege to access the memory access time. This is also needed for the covert channel, and for the performance analysis of the covert channel. Time measurement is the key to launch most popular cache attacks, like Flush+Reload attack, Prime+Probe attack, etc. The attackers collect the time stamps and process them locally to retrieve information. To ensure accuracy, the attacker have the access to consult with several registers. It is possible because TrustZone is not trapping those instructions.

In this chapter, we also assume that the operating system is not compromised so that the attackers are forced to use covert channels or side-channels without explicitly violating access control policies enforced by the operating system or other protection mechanisms. We assume that the attacker has sufficient privilege to access the memory access time. This is also needed for the covert channel, and for the performance analysis of the covert channel.

Chapter 4: DESIGN AND IMPLEMENTATIONS

This thesis includes the design and implementation of a framework based on ARM Cortex-M series. The framework is a security design for defending against several types of malicious attackers. The defense should start from the very first second of the IoT device booting up. The structure of the framework is as follows.

4.1 Vulnerabilities on ARM Platform

In this thesis, we are targeting to the threats of the IoT devices based on ARM platform. Some papers had research on the design and the structure [2]. On ARM, most security design and implementations are targeting to these types of the vulnerabilities:

Broad-level Exploitation. For this type of attacking, the attacker utilizes the possible open debugging ports to downgrade the whole system into a version, which is open to it to attack. Besides, leaving some other things like test points open can also be risky for the probing attackers. Several kernel downgrading exploits are reported, making it a threat. For example, some vulnerability on XBox 360 can be exploited by the attacker to downgrade the system [6]. The system downgraded then exposes a serious weak point to timing attack.

Chip-level Exploitation. For IoT devices, they are designed always with special purposes, with some designated microprocessors and micro-controllers. However, the flexibility often makes the design share some common functions. Although the microprocessors and micro-controllers are designed differently in different devices, the documentation can still leak some common function operations. If an attacker is smart enough to such 'public knowledge', the system is under serious threats based on it. Some papers introduce the memory exploitation on Actel ProASIC FPGA, which can extract the stored AES key [29]. Vendors like Chipworks can performance reverse engineering tasks on IoT devices [31].

Boot Process Vulnerabilities. As the IoT devices often have simple or no OS, the booting of the device can leave vulnerabilities. The attacker can just inject some malicious payloads into the

booting process, as booting of the device is not providing blocks to the malicious injections. There are several reports about the attack targeting to this. Some mobile devices like iPhones can be injected with malicious payloads [14].

Remote Access Channels. For IoT devices and smart devices, the design often has the remote controlling mechanism. As an assist to this, the manufacturer may use some open channels for upgrading and connection. The channels are not always secure, leaving with some risks for being utilized by the attackers. Some malicious attackers even change the firmware into a back-door holder enabling the information passing through [33].

This thesis targets to cache threats on ARM platform. From recent research papers, most of the threats falls into side-channel attack and covert channel attack. For side-channel attack, the attackers utilizes information based on what they gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself. For example, timing, power consumption and noise levels. In this thesis, the side-channel threats are based on cache loading time measurements.

Covert channels, on the other hand, are a different set of threats creating illegal transfer information between processes, which are not allowed by computer security policy. These channels can tunnel in hidden ways, and can be countered by system performance monitors, and some covert channels suffer from low bandwidth. Covert channel attacks are out of scope of this thesis, but our defense can also counter covert attacks in the way cutting down their bandwidth.

4.2 Design Goal

To the security problems issued above, we design the framework in order to try blocking the types of attack getting in touch with the IoT devices. According to previous research papers, the system of Overshadow [7] can provide an isolated and secure running environment for the apps to be protected, or connections to be encrypted. We are using similar design idea, but we are also aiming to make the protection fit for IoT devices. As a summary of the design, we are trying to achieve these three goals.

Flexible, portable but not sharing. This means that the protection can be implemented like a wrapper injected between the firmware and the mini-OS, making it possible to change very little from device to device. On the other hand, the implementation should also be protected, with no need to pass the documents of every single function. The manufacturer just need to port the whole framework to the device, and then it will work. If sharing too much, the risk of the chip-level exploitation can be a serious problem, as we mentioned above.

Safe and secure. The implementation can provide defense against some of the most widely-used attacking styles. The defense should also start from the very beginning of booting, and even protect some ports from recent cold-boot attack.

Low overhead. Power consumption has always been a serious concern to IoT devices, as the developers can never design something that like an OS on PC, or some large parallel programs. Limited by the hardware environment, the security framework should be light weighted, taking up the sources as less as possible.

4.3 State Transaction

As shown in Overview, our defense platform is like a protection layer between the kernel and the hardware. In this section, we discuss the state transaction inside and outside the protection we design.

When the user is executing an application that needs protection, the kernel and our designed handler is working to service system calls or interrupts. The figure below illustrate the working flow. When an application performs a system call in step 1, it triggers the handler to save the contents of the registers to the safe registers that provided by the handler, and the control of the registers are granted to the handler. Then, the registers containing the application's contents are cleaned.

After that, the handler turns to the system kernel to handle the system calls. If the system calls contains with the arguments that can be harmful to the system, the handler intercepts the system call and return with denied information to the application. Otherwise, the system kernel

will respond to the system call.

4.4 Research Challenges and Features

Cache threats are becoming research focus points in recent years. As mentioned in Literature Review chapter, we introduce some research papers about cache-based threats and defense. However, our research and experiments are based on ARM, for multiple reasons. Here are some of the research challenges and features in our research.

Firstly, ARM security is much different than traditional x86 platform system security research. Most of the papers mentioned above are just trying to design and implement a defense model that works for all. However, due to the limitations on ARM in instruction sets and memory design, some of the defense model may not work well, or even worse, cannot apply to ARM platform. Our research work is trying to design and implement defense framework for ARM platform, meaning that we do not port previous design on ARM and make it executable. The defense on ARM is a different topic comparing with x86 system security.

Secondly, security design on ARM is challenging and we must consider many sides. ARM is designed with multiple privilege levels, which is challenging the security design with access control in different privilege levels. There are also very few instructions, like $rdtsc$ that we can use on x86 to simplify our special instructions. We have to go to the assembly code layer and consult some performance counting registers to get what we need for our implementations. Detailed implementations are shown at later chapters, showcasing the difference between ARM and x86 platforms.

Thirdly, ARM products are in a great variety throughout the years. From PC-like platform to mobile phones, and some smart home or smart vehicle devices. Nearly all of them have special limitations and challenges. For example, our test platform on Cortex-M is based on M3 structure, which is with no cache in the chip level. The main memory is fully allocated and cache here is just a 'cachable' block of memory that acts like cache on other platforms. This design is fit for the devices with limited power supply and limited computing power, since they do not cost

performance overhead in paging and locating address. We have to face the challenges on the ARM platforms, and design light weighted defense framework to serious cache threats.

There are some other challenges to our research work. For example, we have to implement side-channel attack on ARM and test both on the attacker side and defense side. When we calculate the performance overhead of some instructions, we need to find some accurate and commonly used benchmarks.

Our research, with the challenges on the way, is also helpful in future defense design and implementations. The data we collect about performance overhead of some instructions can give upper and lower bound of defense framework based on those instructions. On the other hand, our research provide ways of utilization of TrustZone, which is also a design feature for ARM platform.

4.5 Design Features

On devices with ARM chips, security design can be quite different from the same case on Desktop or even mobile phones. We even have to think about the difference with traditional design on ARM utilizing TrustZone. In this part, we analyze our design features, challenges and show how our design fit for the new ARM devices.

According to our design goal, we need to make sure the security framework we design is flexible. It should be easy to port from device to device, despite the function or the use of each device. For example, if the secure handler we design and implement is porting from a smart home monitor to a series of smart vehicles, we have to ensure the manufacturer is doing as little as they wish to make the system fit in to the new environment. On Desktop and other PCs, it is relatively easier because of the standard OS and capsuled interfaces. However, on ARM devices, we get very little from the OS, so we have to implement the flexibility within our framework.

The next critical issue for the ARM devices is power consumption. With the consideration of that, we have to discuss the need of the presentence of TrustZone once again. Although doing every implementation in TrustZone is simple and easy, it is not the best energy-efficient solution

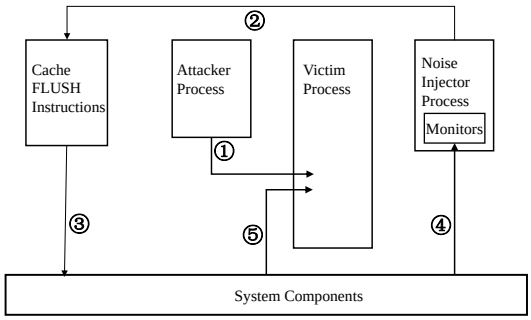

Figure 4.1: Defense Model Overview

sometimes. To this target, we try to use the privilege level of ARM to work like TrustZone and thus cut down the energy cost. Energy is not a serious problem in the devices like smart home devices and smart cars, as they can easily recharge. However, it is a problem in some other devices like outdoor devices and wear-on devices. This makes it another challenging part of our design and implementation. The paging difference is also a challenge to our work.

Given the design of the project, we do not depend on the Hypervisor mode of ARM structure, and not rely on TrustZone protections.

Overview of our design is shown at Figure 4.1. In this figure, we use 1 to 5 to indicate different steps of a side-channel attack and the defense we design against to it. An attacker can utilize the cache to launch side-channel attack, i.e. Flush+Reload attack, shown as step 1. To effectively defend against the side-channel threat, we use Flush injection to cut down the bandwidth of the side-channel. On step 2, the noise injector sends cache FLUSH request, and connect with system components on step 3. Then, the cache is FLUSHed as step 5, and send some performance parameters to the monitor in noise injector as step 4. After the whole loop, the monitor can decide whether the injector should send some other requests, based on some data collected. On the next section, we introduce the design of the monitor, which is shown at Figure 4.4.

4.6 Process Structure on Cortex-A Platform

As mentioned above, the very first step for our experiment is to calculate the cost of entering and exiting from the TrustZone. On ARM Cortex-A Platform, an instruction smc is used for connecting

the secure world and non-secure world. While in normal non-secure world, some code could call privileged smc instruction. Then, secure world monitor will be triggered after validation. After execution of secure code, the return of the execution also calls smc to get back to the normal world. There are many open-source test platform to measure the world switch latency, and in this experiment, we use the well-known QEMU to test. It had been developed since the first patch published in 2011, and been patched by many manufacturers including Samsung, utilizing ARM TrustZone for security design.

The process structure is show at Figure 4.2. When there are smc instructions trigger the TrustZone entry/exit, we trap the instructions and start using perf and other time measurement tools to calculate clock cycles they take to finish switching between trust environment and outside memory. We also FLUSH cache every time when we exit from TrustZone and see the difference in performance overhead by different frequency of TrustZone related instructions.

4.7 Process Structure on Cortex-M Platform

On ARM-v8 platform, SG/BXNS instructions are used to enter and exit from TrustZone. As there were almost no proper TEEs for ARMv8-M on the market as we were testing, we use a testing program instead. SG (Secure Gate) instruction is called by non-secure world code that wants to trigger TrustZone protection. Unlike Cortex-A structure, on ARMv8-M, the page table is not used, so the memory is fully mapped with different regions. When SG instruction is called, the reserved regions for secure world are used to execute the protected part of the code. After the secure execution within TrustZone, the code has an exit called BXNS/BLXNS (Back to Non-Secure) that can lead the execution to other region besides protected ones by TrustZone. We make use of the mechanism of this, and the structure of the testing program is as Figure 4.3 shows.

The term 'cache' here on ARMv8-M is part of normal memory being set as 'cacheable'. In other words, it is a region set aside for possible cache using. On Cortex-A series chips or x86 chips, cache flush operations are just some instructions with privileges. However, the case are different on ARMv8-M. The allocation of a memory address to a cache address is defined by the designers

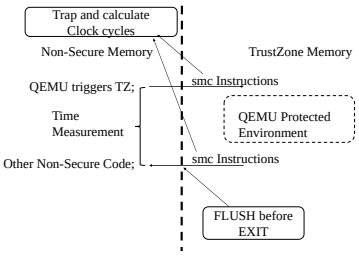

Figure 4.2: Process Structure on Cortex-A

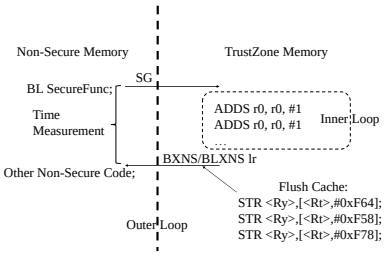

Figure 4.3: Process Structure on Cortex-M

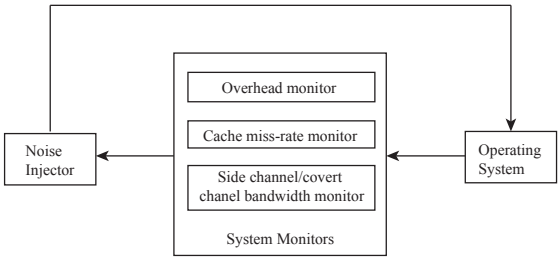

Figure 4.4: Adaptive Control Design with Monitor.

of the applications. Because of the special structure of ARMv8-M, the cache FLUSH operations are sets of DSB (Data Synchronization Barrier) operations, with address-related instructions.

4.8 Adaptive Control Model

Based on the design of defense model using FLUSH operations introduced above, we must find some balance between bandwidth elimination and performance overhead. As sometimes we need better performance and ignore minor bandwidth effects, we need to have some adaptive and flexible controlling methods to keep the balance of performance and security. As a result, we design a monitor between the noise injector and the OS. The designed architecture is shown on Figure 4.4.

For the monitor, we can set up different parameters of each category, and decide the FLUSH requests to the injector.

4.9 Implementations

In this section, we introduce some implementation details in the defense. On ARMv8 with TrustZone, the noise injector can send cache operation requests to the system, and system components

can handle them and FLUSH cache for their needs. We control the frequency of FLUSH operations, and thus keeping a balance of security and performance.

For the structure we implement, the most critical parts are accurate time stamp collection, and cache FLUSH operation. For the non-secure world, as the users do not need to change their non-secure code, we do not need special care of them. For accurate time recording, it is needed for the analysis of bandwidth, and performance overhead. For cache FLUSH operation, it is the key to ensure the cache not going to be utilized by attackers.

4.9.1 Cache FLUSH Operations on ARM Platform

In order to add noises into the message channel, we consider having additional cache FLUSH operations. We use the third process to randomly add cache FLUSH operations, which do not target at some specific programs. As a result, these noise injections can be considered to protect the whole system. Implementing the FLUSH operation on x86 platform is straightforward using $clflush$ instruction, but more complicated operations are on ARM platform for that.

As discussed above, on ARM, users have no access permission to cache FLUSH operations, as these operations are at privileged level. To better researching on the defense strategy on ARM, we build a message channel across the processes. However, on ARM platform, we do not have the instructions like clflush on x86 that are straightforward to deal with cache FLUSH operations. As a result, we have to take a look at the cache allocation on ARM, and use inline assembly codes to implement cache FLUSH operations.

There are two types of cache allocation on ARM: 1) read-allocate cache, and 2) write-allocate cache. When operating on cache with WRITE, if the cache is missed, CPU will simply put the data into main memory. Pre-fetching happens only with READ operations. However, when writing data with cache misses, pre-fetching will happen and CPU will read the corresponding places in cache and start to WRITE. On ARMv8 architecture, the processor uses C7 register of CP15 to implement cache and write buffer operations.

It is usual to clean the cache before flushing it, so the external memory is updated with any

dirty data. The following code segment shows how to clean and flush the entire cache.

```
MOV r0, #0 ;Clear R0;
MCR p15, 0, r15, c7, c10, 3 ;
//Flush DCache;
```

On ARMv7 or higher, the cache FLUSH operations that are privileged and can be handled by ARM. In the code above, we can see the assembling code of flushing the cache uses `MCR` (Move to Coprocessor from Registers). The privileged operation using this can be trapped by the system, and system handles the operation referring to it. The reason is that, when an instruction uses `MCR` or `MRC`, the registers CP14 and CP15 are taken access. These registers are designed by ARM with special purpose, and used only for cache maintenance. For ARM, it has a system call which takes an array of those operations each specified by the struct called $mmuext\_op$. This call allows access to various operations which must be performed with privileged level, like TLB operations, cache operations, and loading descriptor table base addresses.

4.9.2 Time Measurement on ARMv8

Unlike performance counters on x86, on ARMv8 platform, there are no instructions like perf to collect time-related performance counters from the system layer. Another challenge is that we cannot use rdtsc instruction to get time stamps as we often do on x86. Additionally, some other coarse-grained way like $gettimeofday()$ certainly does not work.

Given these limitations, we have to be back to hardware, and look at ARM structure itself. We look up ARM whitebook and find some registers that we can retrieve time stamp information. However, when consulting with these registers, we have to enable them from kernel mode. By default, the access to these registers are disabled.

The following code segment shows the instructions for calculating time:

```
ISB; MRS %0, cntvct_el0;
  //process execution;
ISB; MRS %0, cntvct_el0;
```

Table 4.1: PMU Events on ARMv8 Cortex-A

Event Number	Event mnemonic	Description
0x0001	L1I_CACHE_REFILLa	Level 1 instruction cache refill
0x0003	L1D_CACHE_REFILLa	Level 1 data cache refill
0x0004	L1D_CACHE	Level 1 DCache Access
0x0032	LL_CACHE	Last Level data cache access
0x0033	LL_CACHE_MISSa	Last level data or unified cache miss

```
ISB; MRS %0, cntfrq_el0;
```

We store the timestamps in two arrays and calculate the time based on these raw data. The instructions are privileged, and we can use timestamps for many monitor jobs. $cntfrq\_el0$ is used for reading current running frequency, which is not always the CPU frequency or clock frequency.

4.9.3 Monitors Setup

On Overview section, we introduce the structure of adaptive defense design. It is critical for the defense to have proper monitors in order to provide accurate performance and overhead information feedback data. The challenge of the work is the difference between platforms of ARM and x86. On x86 chips, some system tools like $perf$ can directly present what is going on to the system. On ARMv8-A, however, we have to look up for right registers and use MRS or MSR instructions to read out system performance data. After that, we use some calculations to show the conditions of performance overhead, cache miss rate and bandwidth.

Performance Monitor Units ARMv8-A structure provides various Performance Monitor Units (PMUs) to store system running condition data. In our implementations, we basically use instructions MRS and MSR to collect data from register $PMEVCNTR0\_EL0$, which is a 32-bit performance monitor counter register, and register $PMEVTYPER0\_EL0$ register, which is used for setting up events to be counted. With events shown as Table 4.1, we can calculate current cache miss rate.

Performance Benchmark For ARMv8-A series, ARM has a set of benchmark tools called CoreMark [18]. This benchmark is open-source and fit for features on ARM platform. During the defense, we can modify CoreMark to report performance overhead, in order to work as a monitor that supports adaptive feedback to noise injections. In particular, we modify $core\_matrix.c$, $core\_state.c$ and other source files to report current overhead of the system with running defense.

Bandwidth Monitor Similar to cache miss rate calculation, we cannot directly find data from registers to have bandwidth feedback. However, when we are executing victim and attacker processes, system read and write rates are also showing in different values. As a point of work, we utilize the system read and write rates to calculate bandwidth. Other references for the calculation are the frequencies of the core Cortex-A53 and Cortex-A57.

With the help of PMUs, benchmark tools and related instructions, we can setup adaptive monitors to watch the performance, bandwidth and cache miss rate information of the system. In fact, with more PMUs being utilized, we can setup more monitors to have better control over the noise injections. These can be some additional tasks to work on in the future.

4.9.4 Side-Channel Attack Implementations

In this thesis, we also have implemented test cases of side-channel attack on ARM platform, targeting to $libjpeg$ and some image file. In real life, shared libraries like $libjpeg$ are always the target to be attacked by malicious users, since they can be utilized to retrieve image which contains information that is easy to read. On the other hand, when we try to use noise injections into the message channel, the attackers get low quality images with noises, but they can still retrieve enough information from the picture if it's still readable. Sometimes they can even use Error Correction Code (ECC) to try fixing their results. ECC code can fix images with low level noise injection, but perform worse when we inject too much noise that exceeds the threshold based on the signal/noise ratio.

The side-channel attack we implemented are recurrence of the experiments in the paper by Ruby Lee and his team [19]. We ported to ARM platform and also inject noise into the process

of this attack. Our attack tests are based on Flush+Reload attack, and the results are shown and discussed in Evaluation chapters.

4.9.5 Other Implementations

Besides these, we also have other implementation features on this defense framework.

We use Error Correction Code (ECC) to try recovering the contents missed due to quality loss. In this thesis we use CRC code to work as a checking and correcting process to try recovering the message that the sender just puts into the message channel. That is possible because the attacker may use some ECC to recover the data.

CRC is widely used in digital networks, and storage devices to detect abnormal data due to accidental changes to original data. At CRC, data are packed into blocks with a short check value attached, based on the remainder of a polynomial division of the contents of each block. CRC is popular in network applications because it is simple to implement, easy to analyze the data package from the check value, and good at detecting noise in message transmission channels.

However, CRC and other error correction codes have limitations. When we inject noise beyond a threshold, error correction may not work well, with some cases even performing worse and cannot correct the message according to the checksums. In our experiments, we add much noise into the message channel to defend against the attacker. As a result, CRC performs not well when the noise is injected for too much. It supports the noise injection mechanism for effective defense, as the attacker cannot even use ECC to recover original data.

We also implement a loop to control the frequency of FLUSH operations. The frequency is decided based on the performance monitor. Therefore, the total amount and frequency of noise injection are controlled in the protection side.

Chapter 5: EVALUATION AND DISCUSSION

5.1 ARM Instruction Cost Experimental Results

5.1.1 Cost of Entering and Exiting from TrustZone on Cortex-A

QEMU with ARM TrustZone provides us a variety of tests. The tests behave as we users initiating secure operations from user mode. The test functions validate the TrustZone features of QEMU, and utilizing the features of the functions themselves. We have tests on read/write from non-secure world to secure world and vice versa. The results are shown as Table 5.1 shows.

5.1.2 Percentage of TrustZone-Related Instructions

We write a script based on the above write/read code. In the script, there is a loop called in and runs several times as a workload. We use Ubuntu 16.10 as the normal world OS, with 26 processes running on background, including the workload we use for testing. We count the smc-related instructions that belongs to TrustZone-related operations, and analyze the attributions of them. According to our test, the instructions takes up less than 6% of the total instructions running, with these three different categories as shown on Table 5.2.

In normal using conditions, however, the manufacturers are not using TrustZone that often. Thus, the test here can be the upper bound or 'worst case' of the utilization of TrustZone-Related instructions. Normally, the non-secure world does not have to call in the secure world too often.

5.1.3 Performance Overhead by FLUSH Operations

It is already known that ARM TrustZone on Cortex-A series are not going to clean the cache when exiting from the secure world to non-secure world. As a result, there are possibilities for the

Table 5.1: TrustZone-Related Instruction Cost on Cortex-A

Tests	Direction	Average cost (Clock Cycles)	Time on 800Mhz
P0_nonsecure_check_register_access	Non-secure to Secure	1950	2.43us
P0_secure_check_register_access	Secure to Non-secure	2200	2.75us

Table 5.2: Different Categories of TrustZone-Related Instructions

Type	Percentage
Non-secure to Secure Test R/W	2.87%
Secure to Non-secure Test R/W	2.91%
Others (Access from Background)	0.01%

attackers to make the most of the last level cache and conduct cache-based attacks. For example, the side-channel attack of FLUSH+RELOAD, PRIME+PROBE are both found practical on the environment with TrustZone on ARM Cortex-A, some even with a fiercely high bandwidth. On the other hand, if we can FLUSH the cache every time on the 'exit' to the normal non-secure world, then it can be expected that the bandwidth of the side-channel attack can be limited to a number that is worthless to the attackers to gather the information possibly leaked by the smc operations.

We still test the performance using our test model. In this test, we are adding cache FLUSH operations on every smc instruction that calling exit from the secure world to non-secure world. On that situation, we measure the performance overhead by comparing the clock cycles of execution. At the same time, we change the percentage of TrustZone-related instructions to see the difference in the overhead. The results are shown on Figure 5.1 and Figure 5.2.

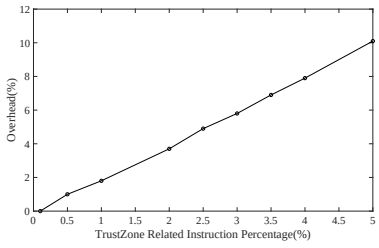

Figure 5.1: TrustZone Related Instructions and Their Overhead

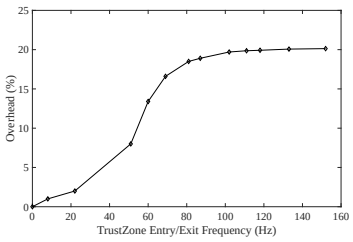

Figure 5.2: TrustZone Entry/Exit Frequency and FLUSH Overhead

Table 5.3: TrustZone-Related Instructions Cost on ARMv8-M

Operation	Direction	Cost on Average (Clock Cycles)
SG	Non-Secure to Secure	3.5
BXNS/BLXNS	Secure to Non-Secure	5.2

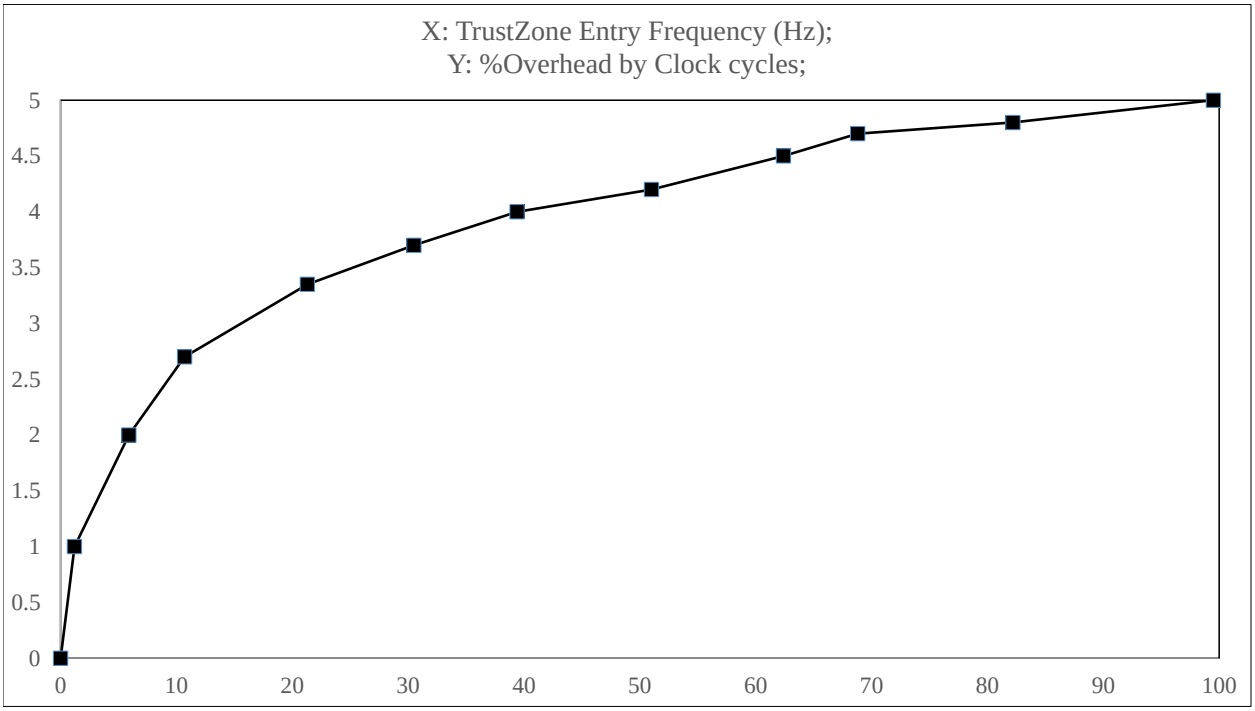

Figure 5.3: TrustZone Entry/Exit Frequency and FLUSH Overhead on ARMv8-M

5.1.4 Experimental Results on ARMv8-M

According to our experiments, the testing case triggering TrustZone operations SG and BXNS. As every region is fixed in the memory, the costs of entering and exiting from TrustZone are surprisingly much lower than ARM Cortex-A series chips. The results are shown at Table 5.3.

We measure the performance of the FLUSH operations using our testing program shown at Figure 4.3. We add FLUSH operations before executing BXNS/BLXNS operations to ensure there is nothing left when exiting from TrustZone. We measure the overhead by the FLUSH operations, and we also change the outer loop to have different frequencies of TrustZone entries and exits. The results are shown at Figure 5.3.

5.2 Experimental Setup on Defense Based on FLUSH Operations

In this thesis, we have different sets of experiments, testing the effectiveness of Flush-based adaptive defense. According to the experimental results, we have discussions on ARM Cortex-A series.

For Experiments, we target on two core problems: TrustZone and cache threats. For TrustZone experiments, we have experiments on following aspects:

- Percentage of TrustZone-related instructions;
- Cost of entering/exiting TrustZone;
- Effectiveness of TrustZone by bandwidth.

For cache threats, the major threat we focus on this thesis is side-channel attack. We have experiments on the following aspects:

- We FLUSH cache while exiting TrustZone and test the effectiveness;
- Cost of FLUSH operations;
- Effectiveness of FLUSH operations by bandwidth.

We also have theoretical discussions based on the experimental results. We have three aspects of theoretical analysis:

- We discuss bandwidth effect of FLUSH operations by theory;
- We discuss overhead effect of FLUSH operations by theory;
- We discuss defense performance by entropy.

For the first two aspects of discussion, we use curve regression to match the experimental results and theoretical discussions.

We evaluate our proposed defense mechanisms using a proof-of-concept implementation on ARMv8 Platform. On ARM, we use a Juno r1 Development Platform, with one A57, one A53, the cache of L1 48KB for instruction, 32KB for data, and L2 for 2MB.

Table 5.4: TrustZone-related Instructions Cost on ARMv8 Cortex-A

Tests	Direction	Average Cost (Clock Cycles)	Time Cost on 800Mhz
P0_nonsecure_check_register_access	Non-secure to Secure	1950	2.43us
P0_secure_check_register_access	Secure to Non-secure	2200	2.75us

Table 5.5: TrustZone-Related Instruction Count

Type	Percentage
Non-secure to Secure Test R/W	2.87%
Secure to Non-secure Test R/W	2.91%
Others (Access from Background)	0.01%

5.2.1 Cost of Interaction with ARM TrustZone

On ARM Cortex-A Platform, an instruction smc is used for connecting the secure world and non-secure world. While in normal non-secure world, some code could call privileged smc instruction. Then, secure world monitor will be triggered after validation. After execution of secure code, the return of the execution also calls smc to get back to the normal world. There are many open-source test platform to measure the world switch latency, and in this experiment, we use the well-known QEMU to test. It had been developed since the first patch published in 2011, and been patched by many manufacturers including Samsung, utilizing ARM TrustZone for security design.

QEMU with ARM TrustZone provides us a variety of tests. The tests behave as we users initiating secure operations from user mode. The test functions validate the TrustZone features of QEMU, and utilizing the features of the functions themselves. We have tests on read/write from non-secure world to secure world and vice versa. The results are shown as Table 5.4 shows.

We also write a script based on the above write/read code. In the script, there is a loop called in and runs several times as a workload. We use Ubuntu 16.10 as the normal world OS, with 26 processes running on background, including the workload we use for testing. We count the smc-related instructions that belongs to TrustZone-related operations, and analyze the attributions of them. According to our test, the instructions takes up less than 6% of the total instructions running, with these three different categories as shown on Table 5.5.

In normal using conditions, however, the manufacturers are not using TrustZone that often.

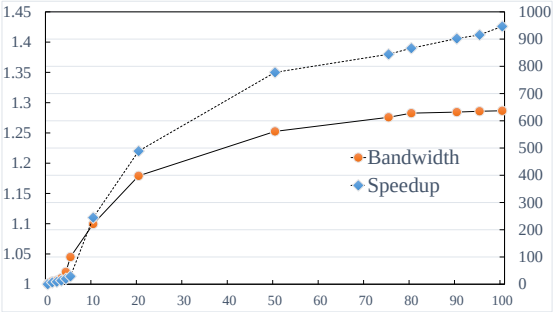

Figure 5.4: Bandwidth of a Flush+Reload based side-channel and performance improvement by allowing a specific ratio of cache operations passing through ARM handling.

Thus, the test here can be the upper bound or 'worst case' of the utilization of TrustZone-Related instructions. Normally, the non-secure world does not have to call in the secure world too often.

Noise Injection to FLUSH+RELOAD based Side-Channels on ARMv8

As we described in Overview, on ARM platform with TrustZone protection, ARM can provide some protection for the cache against side-channel attack on the cache, using cache invalidation. However, it introduces significant performance loss. We revised implementation such that the amount of cache invalidation is under our control. Figure 5.4 shows the experimental results for a `Flush+Reload` based side-channel.

In the figure, x-axis is the percentage of cache operations passing through the system handling. In other words, it is the percentage of cache operations that run on the processor hardware directly rather than being ignored. y-axis on the left is the performance speedup and y-axis on the right is the bandwidth of the side channel. If 0 percent cache operations passes through ARM handing, the bandwidth of the side channel will be 0 and we set the corresponding execution time as the base value for the speedup measurement. When we have more cache operations passing through the ARM handling, we can get better performance on the execution. However, the bandwidth of the side channel goes up as well. According to the experimental results, when we increase the percentage of cache operations passing through ARM handing, the bandwidth of potential side-channels in the whole platform will increase quickly up to 650 bits/second which is very practical and useful for side-channel attacks. At the same time, we can have performance improvement

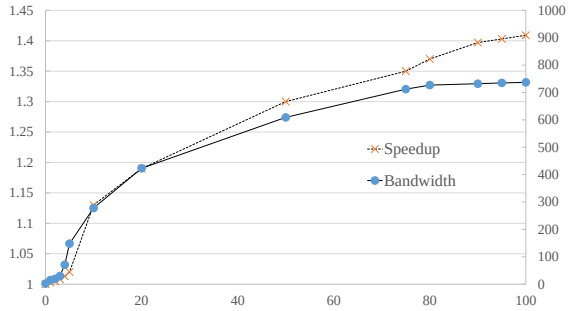

Figure 5.5: Bandwidth of a Prime+Probe based side-channel and performance improvement by allowing a specific ratio of cache operations passing through ARM handling.

up to 43%. However, we do not want such kind of performance improvement due to the security risks of side-channel attacks. The trade-off must be chosen between a non-practical bandwidth of side-channel and acceptable performance improvement.

Noise Injection to PRIME+PROBE based Side-Channels on ARMv8

Similarly, when we inject flush operations to a system, it can also affect the bandwidth of PRIME+PROBE based side-channels. In our experiments, when we inject flush operations to incur about 20% overhead, the bandwidth of a side-channel can be decreased from about 600 bps to only several bps. The flush operations can effectively interfere with the time measurement in PRIME+PROBE based side-channels, thus making it non-practical.

Figure 5.5 shows the experimental results of noise injection into a PRIME+PROBE based side-channel. In the figure, x-axis is the percentage of cache operations passing through ARM handling. y-axis on the left is the performance speedup and y-axis on the right is the bandwidth of the side channel. The configuration of the experiment is the same as the FLUSH+RELOAD based side-channel except the type of side-channel is PRIME+PROBE. In the figure, we can see the different impact on speedup of the running of the program and bandwidth of side-channels caused by different percentage of passing-through cache operations from the injector process. When we pass-through more cache operations of a process (not trapped by the ARM platform and invalidate cache lines) we can see the speedup of an application increases, but also with increasing risks of side-channel attacks, as shown by fast increasing bandwidth of side-channels.

On ARM, as shown in Figure 5.5, when the ratio of not trapped cache operations increases to be about 10%, the bandwidth of side-channels quickly rises up to more than 200 bps, with speedup rising for only 13%. When the ratio increases to 75%, the bandwidth of side-channels rises up to more than 700 bps, with the speedup of application only by 35%. As a result, we can see that enabling ARM to pass-through some cache operations is not affordable, with very high risks of leaking information through side-channels. In other words, the way ARM handles the cache operations by processes is necessary to ensure security given the performance overheads. Otherwise, there will be greatly increased risks to have side-channel attacks on ARM platform.

5.3 Other Experimental Setup and Results

In this section, we have some experiments of our implemented framework, and some results of these experiments. The experiments are in three parts. In the first part, we use a sender and a receiver passing message through a channel, and try stealing information based on the memory access time. This experiment is carried at x86 platform, with another process inserting randomly FLUSH operations into the channel to inject noise into the message channel as a defence strategy. We are counting the bandwidth of the message channel, and other things like the entropy impact by additional noises. In the second part, we are using ARMv8 platform, and use another set of instructions to insert FLUSH instructions. We adjust the way of handling cache FLUSH and see the difference of performance and bandwidth based on this.

We evaluate the performance of the covert channel on both x86 and ARMv8 Platform. On x86, we use a computer with an Intel® i7-4790 CPU at 3.60GHz and 16GB RAM. The operating system is Ubuntu 14.10. On ARM, we use a Juno r1 Development Platform, with one A57, one A53, the cache of L1 48KB for instruction, 32KB for data, and L2 for 2MB.

5.3.1 Experiments on x86

On Figure 5.6, the results show the impact of FLUSH operations on the bandwidth and performance. The running time of the whole process can be collected using $perf$. We use a bitmap

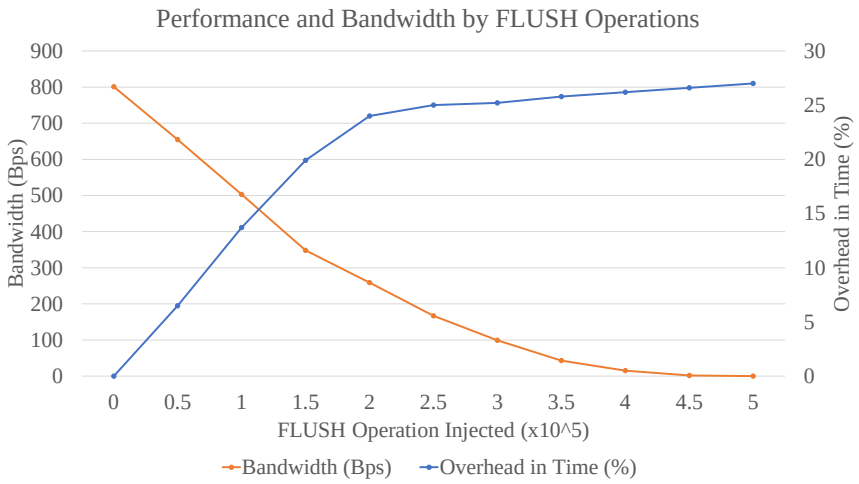

Figure 5.6: Experimental Results on x86

image as message passing through the message channel, and the noise injection can affect the image that the possible malicious side-channel can get. From the Figure, we can see the frequency of FLUSH operation inserted has impacts on both bandwidth and performance. Comparing with the situation without FLUSH operations, the performance overhead ranges from around 1% to around 30%, while the bandwidth of the side-channel can lower down from several hundred bps to less than 30 bps.

For each test case, we also try using CRC to restore the noisy file that the receiver gets. However, CRC or other error correction cannot perform well when the noise being injected for too much. Sometimes, these error correction mechanism even get worse results. Figure 5.7 shows one of the results. On that case, we had added 50k FLUSH operations into the message channel, comparing with a bitmap of 172kb size. As a result, we can get a noisy bitmap file from the receiver like the second picture, and CRC performed badly this time, forming out the result as the rightmost picture.

Another type of data we had collected is accuracy, which means the accuracy for the receiver to correctly 'guess' the right corresponding bits. This can be used to measure the entropy of the channel, as shown above in this thesis.

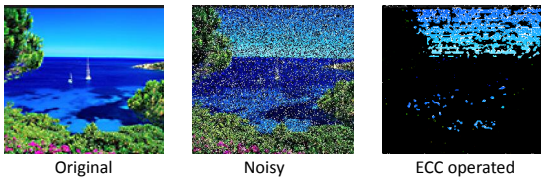

Figure 5.7: Noisy and Error Correction

5.3.2 Side-Channel Experiments on ARMv8

On Figure 5.8, the results show the impact with the system handling with cache FLUSH operations. We change the percentage of FLUSH instructions, and receive different level of the channel performance and bandwidth. Here we are using in-line assembly code to record time in order to receive relatively accurate running time for each test case.

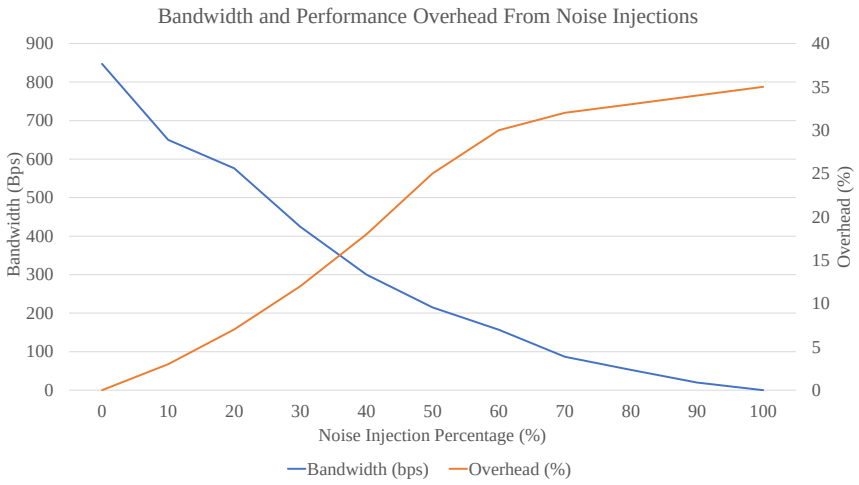

Figure 5.8: Bandwidth and Speedup by FLUSH Instructions

From the results, we can see the increasing bandwidth with the allowance to not trapping the instructions. With more cache FLUSH instructions ignored by the system, the bandwidth of the covert channel is increasing quickly, from nearly 0 to about 600 bits per second. On the other hand, because the trapping activities are cancelled with some percentage, the total running time for the transmission is increasing based on this. We can have a performance of around 45% when

all trappings are cancelled. However we cannot do this, because at this situation the system is losing control of the cache and memory access, leaving a highly risky flaw to the attackers. In this section, we introduce our experimental results and discussions, both on ARM Cortex-A and Cortex-M platforms.

5.4 On the Cost-Effectiveness of ARM TrustZone and Related Instructions

5.4.1 TrustZone Usage Frequency and Flush Overhead

According to our experimental results, on ARMv8 platform, the system is connecting with TrustZone with very low frequency, taking up less than 10% of the instructions at most. Some specific instructions trigger the secure gate of TrustZone. However, when the contexts running in secured memory finish, TrustZone does not clean the cache before exit, leaving some risks here. Based on low frequency and overhead from TrustZone related instructions, we can FLUSH the cache every time when exiting from TrustZone, and still keep a low overhead of less than 20% on Cortex-A chips. This design will let the system manufacturer to put protected or private contexts into TrustZone and with no worries about side-channel attack when exiting from it.

5.4.2 TrustZone Discussion on Cortex-M

Unlike Cortex-A series, ARMv8-M based on Cortex-M structure is designed to have low energy cost and with much simpler system, which is thought to fit for mobile or home devices. At this case, the performance overhead brought by security protection should be controlled in a very low number. According to our experimental results, on Cortex-M structure, the secure gate instructions take much less clock cycles to execute, making it a good choice on the basis of security design. When we add FLUSH operations on exit instructions, we have even lower overhead comparing with Cortex-A chips, having less than 10% overhead at most. It is a practical design for the manufacturer to introduce and not hard to develop. On the other hand, they could put protected data and instructions into the secure enclave of TrustZone.

5.4.3 Cache Based Defense on ARM Platform

Though we have no perfect way to take the place of validating cache and cleaning the TLB entries, we still have some idea for possible solutions, because there are some potential for speeding up and getting better performance. For example, we can move the FLUSH operations out from the privileged level, and try implementing another framework to ensure the security of this type of operations, while maintaining low overhead. In this thesis, we quantitatively discuss the security design for dealing with FLUSH operation requests, and there are still some more topics to research on.

5.4.4 Side-Channel Experiments on Cortex-A

On Figure 5.2, the results show the impact with the system handling with cache FLUSH operations. We change the percentage of FLUSH instructions, and receive different level of the channel performance and bandwidth. Here we are using in-line assembly code to record time in order to receive relatively accurate running time for each test case. From the results, we can see the increasing overhead with the increase of FLUSH instructions. With less cache FLUSH instructions added into the system, the bandwidth of the covert channel or side-channel is increasing quickly. We may inject less FLUSH operations for performance concerns, but we cannot do this in real tests, since at this situation the system is losing control of the cache and memory access, leaving a highly risky flaw to the attackers.

We also use image files for the test. At that time, the noise in the picture increases with more percentage of FLUSH operations injected into the system. Figure 5.9 shows the difference. This is the case when we set up 50Hz FLUSH operations to be added, while another process sending information by the transmission of an image file with 201kb size. We use side-channel strategy to try retrieving the image, in unlimited time (which is not the real-life case since the attackers must act quickly to launch their attacks). We give the attacher unlimited time just to show how serious and practical a side-channel threat can be.

Based on the figure above and experimental results, we can see that with FLUSH operations

Figure 5.9: Noise Injection on ARM

added into the system, the attacker is with more difficulty in retrieving information in the system. Even with little overhead, the defense can be effective. However, the attacker can still have parts of the information, which is another problem of balancing in performance and effectiveness. We have discussions of that in later chapters introducing adaptive defense design based on balance of these parameters.

5.5 Evaluation on Side-Channel Experiments

In this section, we discuss and analyze the experimental results, and have some more theorized discussion on informational entropy and the use of the defense.

5.5.1 Discussion on Experimental Results on x86

From Figure 5.6 we can see the bandwidth and overhead difference by different level of noise injection. From our experimental results, on x86 platform, we can injected randomly FLUSH operations at the overhead with about 15% to 20% and decrease the bandwidth for more than 75%. If a message channel needs more secure environment and does not care the overhead for so much, then we can add FLUSH operations to the overhead of around 40%, and the bandwidth of covert channel can be lower than 1% comparing with the original case. However, the overhead of more than 35% is not a good option to a system, as we cannot spend on the defense against covert channel for that much. Still, additional FLUSH operations can be effective against covert channels.

According to our experimental results, noise injections are indeed effective and practical to

protect against side-channel attacks. One major advantage of such defense is that the noise injection is not specific to an application. The flush of cache protects all applications using the cache and the protection is system wide.

5.5.2 Discussion on Experimental Results on ARMv8

Our second and third experiment configuration produced similar results on ARMv8 structure. From Figure 5.8 we can see the different impact on speedup of the running of the program and bandwidth of covert channel caused by different level of trapping allowance. When we enable some FLUSH operation requests of guest operations to be ignored by the system, instead of trapping in and invalidating cache lines, we can see the speed of the running of the program increases, but with some increasing risks. The bandwidth brought by this increases fast, providing a risky environment for guests.

On ARM, as Figure 5.8 shows, when we enable the percentage of FLUSH operations ignored to be about 10%, the bandwidth quickly rises up to more than 200 bps, with speed rising for only 13%. When the percentage rises up to 75%, the bandwidth rises up to more than 700 bps, with the speed of running only rising for 35%. As a result, we can see that enabling system to 'pass' some cache FLUSH operations is not affordable, with high risks for leaking data and stolen by covert channels.

Though we have no perfect way to take the place of validating cache and cleaning the TLB entries, we still have some idea for possible solutions, because there are some potential for speeding up and getting better performance. For example, we can move the FLUSH operations out from the privileged level, and try implementing another framework to ensure the security of this type of operations, while maintaining low overhead. In this thesis, we quantitatively discuss the security design for dealing with FLUSH operation requests, and there are still some more topics to research on.

5.5.3 Discussion on CRC Correction

As shown at Figure 5.7, we use CRC for error correction in the experiments on x86, and try restoring the picture got from covert channel with much noise injection. However, when the additional noise injected for too much and exceed a threshold, CRC cannot restore the mistaken part, and sometimes even gets worth, like the figure shows.

We use 8-bit CRC for detection and correction. From the mathematical function of CRC error detection, the transmission can detect these four types of errors: 1) any 1 bit error; 2) any two adjacent 1 bit errors; 3) any odd number of 1 bit errors, and 4) any burst of errors with a length of 8 or less.

One of the problems of this correction mechanism is that sometimes it cannot correct the continuous errors with a length longer than N when we use N-bit CRC correction. In our test, when we inject FLUSH operations, the reading of the timestamps can be affected by the injected FLUSH operations. As a result, when the receiver reads in continuous 8 or more bits incorrectly, CRC will fail to correct.

Another problem for CRC is that sometimes it cannot correct with adjacent even number of bits that are incorrect. For example, in some cases, our CRC cannot correct 4 continuous incorrect bits. If CRC handles with this situation, it can just correct the first bit of these 4. When we are testing with frequent FLUSH operations, we can come up with more cases with adjacent 4, 6, or 8 bits incorrect. When this happens, CRC may not handle that correctly.

The result as Figure 5.7 shows can interpret in this way. We are adding in 50k FLUSH operations into the message channel, and sending a bitmap file with the size of about 170 kb. As a result of additional noises, about $\frac{3}{7}$ of all the bits can be incorrect. As a result, the rate for failure in correction can be cauculated as 5.1:

$$\lim_{n\to\infty} \sum_{1}^{n} (\frac{3}{7})^{n} = \frac{3}{4} \tag{5.1}$$

That means, about 3 quarters of the pixels will not be corrected into original bits, just as the

experimental result shows.

On ARMv8, cache FLUSH operations are at privileged level, so we have another view of this problem. We are trapping every cache FLUSH request, clean the TLB by VMIDs and invalidate cache lines. From our experimental results, we find it needed to do all these things, so the design of responding to cache FLUSH operations can be considered as a safe and secure design.

5.6 Discussion

5.6.1 Theoretical Analysis

In this section, we describe theoretical analysis on the quality of the side-channels and also the impact of noise injections.

Information Theory based Analysis

The Shannon entropy [27] of a random variable $X : K \rightarrow \chi$ is defined in Equation 5.2.

$$H(X) = -\sum_{x \epsilon \chi} p_X(x) log_2 p_X(x) \tag{5.2}$$

The entropy is a lower bound of the average number of bits required for representing the results of independent repetitions of the experiment associated with X. In terms of our model, the entropy $H(X)$ is a lower bound of the effective information provided by one bit of the message.

Using our experimental results on ARM as an example, the accuracy at the receiver side with different level of noise injection is shown in Table 5.6. Note that we are using *noise ratio* as a parameter, which is a ratio of flush operations compared with all cache operations.

We use `srand()` to generate random numbers, so the distribution of the flush operations are of normal distribution. The entropy of the side channel has a relation with accuracy of the bits received through the side channel. Thus, we calculate the entropy as follows.

$$H(X) = -\sum_{i=1}^{n} P(x_i) I(x_i) = -\sum_{i=1}^{n} P(x_i) log_b P(x_i) \tag{5.3}$$

Table 5.6: Overhead and Accuracy on ARM

Noise Ratio	Accuracy	Overhead (%)
0	0.918303	0
0.000010	0.790304	1
0.000100	0.685478	3
0.001000	0.596467	7
0.010000	0.526785	15
0.100000	0.513214	25
0.500000	0.495521	30

Table 5.7: Entropy and Noise Ratio

Noise Ratio	Accuracy	Entropy(H(x))
0	0.918303	0.4079
0.000010	0.790304	0.7409
0.000100	0.685478	0.8983
0.001000	0.596467	0.9728
0.010000	0.526785	0.9979
0.100000	0.513214	0.9995
0.500000	0.495521	0.9999

In our analysis, we set the value of b as 2 to calculate the entropy in bits. Now we consider multiple test cases. In each test, we use an ε to measure the percentage of noise injected in the test. Then, we calculate $H(X)$ and $H(Y)$, which are the entropy of the sender and the receiver respectively. As discussed before, the probability for the sender to send a 0 equals to the probability of sending an 1 for a random message. Thus, we could use the following equations to calculate the quality of message channel with noise injected.

$$H(X) = -\sum_{i=1}^{n} P(x_i)I(x_i) = -\sum_{i=1}^{n} P(x_i)log_2P(x_i) \tag{5.4}$$

$$H(Y|X) = -\varepsilon log_2\varepsilon - (1-\varepsilon)log_2(1-\varepsilon) \tag{5.5}$$

$$H(Y) = -\sum_{i=1}^{n}(P(x_i)log_2(x_i) + \varepsilon - 2\varepsilon P(x_i)log_2(x_i)) \tag{5.6}$$

And we have the results with different noise injections ε, as shown in Table 5.7.

In the table, with a relatively high amount noise added into the side-channels, the entropy rises

Table 5.8: Overhead of Noise Injections

Noise Ratio	Overhead (%)	Entropy	Bandwidth (bps)
0	0	0.4079	675
0.000010	1	0.7409	552
0.000100	3	0.8983	449
0.001000	7	0.9728	251
0.010000	15	0.9979	137
0.100000	25	0.9995	95
0.500000	30	0.9999	6

up quickly. It is close to the max value of 1 with around 50% operations added. When we add more noise, it makes the receiver harder to guess a bit from the sender. Therefore, when we have a probability close to 0.5 to fail, the entropy will have the highest value of 1.

Channel Quality

With different level of noise injection, a side-channel constructed by an attacker can be from highly risky to almost non-threatening. As discussed above, with the random injection of flush operations, the values of message entropy, the bandwidth and overhead are changed accordingly, as shown in Table 5.8. In the table, with the noise injected, both message entropy and overhead increase, while the bandwidth of side-channels decreases quickly. With more noise injected into the channel, it makes the channel filled in with additional noise, the entropy value increases.

As shown by Shannon entropy definition, when the entropy is close to 1, the message channel can be considered as very poor quality. For each bit with two possible values, the expected time of guesses for getting the correct bit is close to 2, which is nearly a situation with random guessing. If we have such kind of message channel, it cannot send meaningful message because of the difficulty for the receiver to get the corresponding bits.

However, the injected noises also have some negative impact on the system, which is shown as overhead in our experiments. There is a tradeoff between performance sacrifice and increasing of security. On ARM platform, we can achieve effective defense using flush operations injected into the system, with the performance overhead of about 20%, to effectively defend against the

side-channel.

Statistical Discussion

Now we consider the bandwidth of side-channels in Figure 5.4 again. In the experiments where we randomly insert flush operations to interfere with the side-channels, the time of injecting noise is randomly distributed. Also, the interval of each pair of operations is randomly distributed. Exponential distribution is usually used to describe the distribution of intervals of a set of statistically independent events. In our experiment, we use it to describe the distribution of injected flush operations intervals. Every time the system flushes the cache, it affects the time measurement of the side-channel attacks. Thus, the bandwidth of possible side-channels is cut down. As a result, the flush operations can affect the bandwidth of side-channels, in the way of an exponential distribution.

When we look at the side-channel bandwidth, another factor we have to consider is the background noise from other running processes in the system. We model the system background noise using a uniform distribution. Therefore, the cumulative distribution function is as follows:

$$F(x,\lambda) = \begin{cases} 1 - e^{-\lambda x} & x \geqslant 0 \\ 0 & x < 0 \end{cases} \tag{5.7}$$

Where λ is the rate parameter of exponential distribution. As we focus on the bandwidth of potential harmful side-channels, there cannot be the situation where x is less than 0. In our figure 5.10, we use 100% as normal running performance. When we insert FLUSH operations, the performance will be less than 100% as some of the resource are used to perform FLUSH operations. As mentioned above, we have to take the background noise into consideration. Thus, we have the function with more parameters as follows:

$$F(x,\lambda) = a(1 - e^{-\lambda x}) + b \tag{5.8}$$

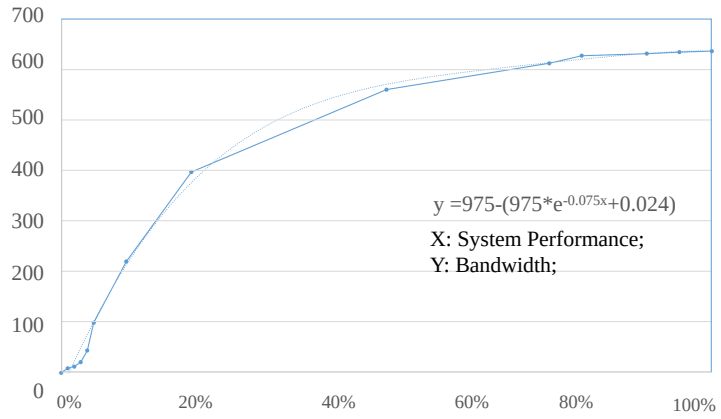

Figure 5.10: A function for side-channel bandwidth prediction based on statistical model.

Where a is the maximum possible bandwidth under our experimental environments and b is the parameters of background noise.

Based on our experimental results and the above statistical analysis, we have a curve fitting function shown in Figure 5.10.

The function with parameters determined by experimental results is as follows.

$$F(x) = 975 - (975 * e^{-0.075x} + 0.024) \tag{5.9}$$

The function is used as a reference for adaptive noise injection and side-channel bandwidth prediction in the defense. On the figure above, on x-axis we use the percentage of theoretically highest performance of the system in running time, and on y-axis we compare the bandwidth of our test with theoretical analysis. In our test case, when we keep the performance of the system as 100%, there are no FLUSH injections added, and the bandwidth can be at highest level. However, when we inject all the FLUSH operations to the system, the performance of the background can be 0% because all the system resources have been used up for noise injections. When that happens, it is certain that potential side-channel attackers are having no bandwidth in the channel because they can get nothing from the side-channel.

The figure and the prediction model give us the theoretically upper bound of the side-channel bandwidth and overhead. From our implementations, the highest bandwidth for the side-channel

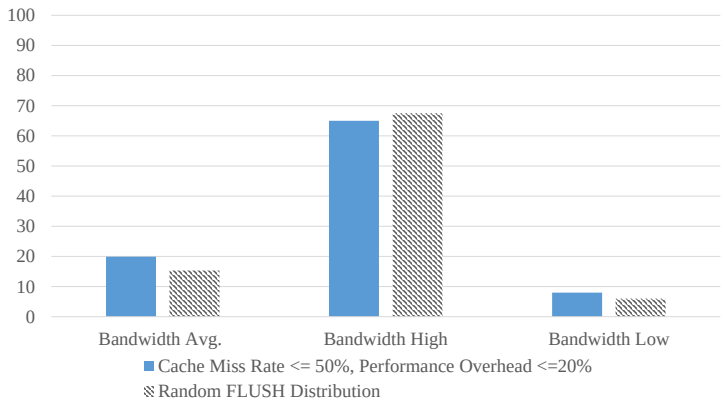

Figure 5.11: Side-channel bandwidth (bps) with and without adaptive mechanism.

attackers on ARM is around 975 bps. The more we add FLUSH operation to the message, the less bandwidth the attackers have to launch their side-channel attack. However, we also observed the rising of overhead is with the noise injections. As a result, we must consider multiple parameters of our defense model, which leads to adaptive noise injection discussions and design.

5.6.2 Adaptive Noise Injection

In the defense against the side-channels, we consider three critical system parameters: performance overhead, bandwidth of the possible side-channel, and the cache miss-rate. In our design, when the performance overhead is over a given threshold, or the cache miss rate is over a pre-determined threshold, noise injection will be stopped to maintain an acceptable performance. However, when the bandwidth of possible side-channels is high enough at a risky level, noise injection will be enabled to protect the system against side-channels.

We have conducted two sets of experiments to compare adaptive noise injection with simple random noise injection. In each set of the experiments, we compare the overhead of the system or the bandwidth of the side-channels. In the first set of experiments, we control the cache miss rate and performance overhead, and see the bandwidth differences between defense with and without the adaptive mechanism. In the second sets of experiments, we control the bandwidth and cache miss rate, and compare the performance overhead between defense with and without adaptive mechanism. The experimental results are shown in Figure 5.11 and Figure 5.12.

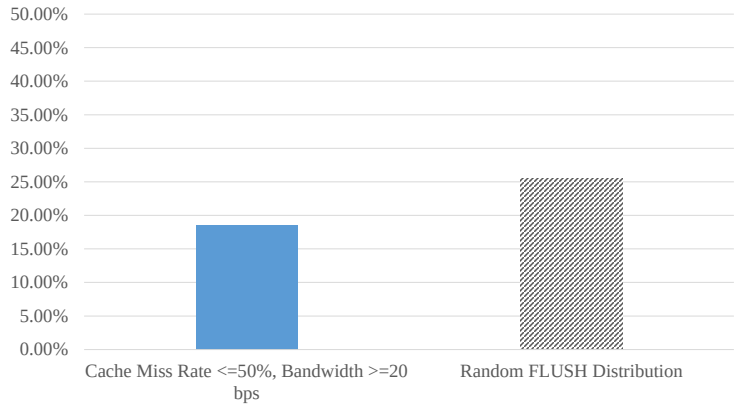

Figure 5.12: Performance overhead with and without adaptive mechanism.

In Figure 5.11, the column on the left shows the experimental results for the adaptive defense. We set up the threshold of cache miss rate to 50%, the performance overhead to 20% and bandwidth to 20 bps. When the cache miss rate is less than 50%, and the performance overhead is less than 20%, we add cache flush operations to interfere with the side-channels. According to the experimental results, when adaptive noise injection is used, the average bandwidth of the side-channels can be similar. However, we can obtain better performance while dealing with high bandwidth situations. When the cache miss-rate is relatively low and performance overhead is low, the risk of leaking information through shared resources is relatively higher. When we target at this situation and inject more noises to the system, the interference can be effective. On the other hand, when the performance overhead is high and the cache miss rate is also high, frequent cache flush operations provide a very tough situation for cache based side-channel attacks. Under such circumstance, there is little need to inject more cache operations as noises. The experiments here show the effect of our adaptive defending.

Figure 5.12, on the other hand, shows the results of the second set of experiments. In this set of experiments, we mainly consider the performance overhead of the defending strategy. Without adaptive noise injection, the overhead is always as high as 20%-30%. However, as some of the cache flush operations are not necessary, our adaptive noise injection can avoid a great amount of flush operations when they are not needed. As a result, when we set the noise injection threshold to the cache miss-rate of 50% and bandwidth of over 20 bps, the overhead average can be optimized

to less than 20%, to be about 18.5%. As we mention the importance of efficiency in defending, this set of experiments prove that we can implement adaptive defending with good efficiency.

We use registers and a loop to work as monitors, which can be more adaptive than trapping 'sensitive' activities. We can change the parameters of the monitors according to our need, with almost no changes on other parts of the defense. It is especially feasible for ARM platform, as mobile devices have different concerns on keeping their own needs to the defense.

When we balance the cache miss rate, security concerns and performance overhead, we can see the difference in our defense platform. In Figure 5.11, comparing with normal noise injection, adaptive strategy has lower bandwidth at worst case but higher bandwidth at the best case. It is because of the balance and the trade-off in performance and security concerns. When the system is under high risks of being attacked, we have to put more resources into the defense, injecting more FLUSH operations. On the other hand, if the system is with less risks of being attacked, high frequency of the noise injections would cost for resources instead of protecting better. At both cases the adaptive strategy would control the frequency of noise injection, thus balancing all the concerns.

At the second set of tests, we can see the performance difference when we use the adaptive strategy comparing with normal noise injections. When we use adaptive strategy to control the frequency of noise injections, we can cost less in performance overhead, returning some resources to the system and to the users.

According to experiments above, when we set up a monitor and control the parameters according to our need, we can have better performance without too much loss on the system's cache miss rate, overhead or security concerns. For further defense design, we can have different parameters fitting into the monitor, and the user can decide which parameters they care most. As a result, the monitor can make the defense adaptive, while keeping FLUSH injections effective.

Chapter 6: CONCLUSIONS

In this dissertation, we have some discussion on the threats, attack and defense based on ARM Platform. We start from x86 structure, investigating the covert channel and side channel attack. Then we port the attack model to ARM platform, both on ARM Cortex-A and ARMv8-M platform. It is shown that the side-channel attack and other types of exploitations are practical and serious, causing loss to users' privacy and security. We try to use Flush and other cache-based defense on these platforms, and with some adaptive ways to cut down the attackers' bandwidth of the channel, while maintaining good performance and lower overhead. From our experimental results, TrustZone can be utilized to help defending against side-channel and covert channel attacks, but it must have an adaptive ways to manage cache operations.

6.1 Future Directions

For future work, we have both theoretical and experimental tasks. On theory research, we have those three categories:

- Study adaptive control method in theory to match the experimental results, and predict the optimal solution of best adaptive control in defense.
- Investigate entropy theory based on experimental results, predictions and related theory.
- Discuss performance of implemented defense framework in theory, and try to have theoretical conclusion on defense against cache-based attack.

On Design, implementation and experiments, we have those three categories:

- Design and implement a defense framework based on ARMv8-M. The overall structure is as Figure 6.1 shows.
- Test the performance of defense framework using some benchmarks, and optimize the framework to good effectiveness and lower overhead.

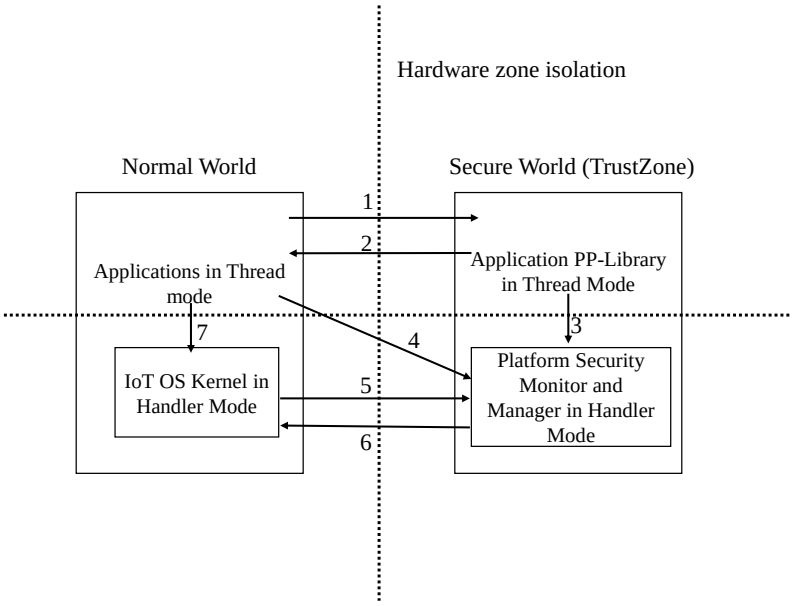

Figure 6.1: TrustZone Entry/Exit Frequency and FLUSH Overhead on ARMv8-M

- Port defense framework to new ARMv8-M boards: M23 and M33 series chips.

For the defense Framework, we will implement critical methods as shown in Figure 6.1 as numbers 1 to 7. It shows the steps for interactions between normal insecure world to the TrustZone-protected secure world. Basically, we would like to introduce access control mechanism to the interactions. When instructions of transferring triggered (steps 1, 2, 4, 5, 6), access control management takes actions to secure related instructions, and perform other instructions protecting the system from side-channel threats. At step 3, security monitor and manager in handler mode control the user's access within the secure world. Similarly, in normal world, we can set some of the OS kernel into handler mode, controlling user's access to some triggering instructions, and connect to the secure world based on the need for security and privacy. Comparing with traditional defense platform based on TrustZone, this platform can be used in more situations and users are not expected to have a long wait every time getting into and exiting from TrustZone.

BIBLIOGRAPHY

[1] Julien Amacher and Valerio Schiavoni. On the performance of arm trustzone. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 133–151. Springer, 2019.

[2] Orlando Arias, Kelvin Ly, and Yier Jin. Security and privacy in iot era. In *Smart Sensors at the IoT Frontier*, pages 351–378. Springer, 2017.

[3] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI'14, 2014*, pages 267–283, Broomfield, CO, October 2014. USENIX Association.

[4] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[5] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge aes against cache-based software side channel vulnerabilities, 2006.

[6] Nancy Cam-Winget, Ahmad-Reza Sadeghi, and Yier Jin. Can iot be secured: Emerging challenges in connecting the unconnected. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2016.

[7] Xioaxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffery Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *In ASPLOS*, May 2008.

[8] Jeroen V. Cleemput, Bart Coppens, and Bjorn De Sutter. Compiler mitigations for time attacks on modern x86 processors. *ACM Trans. Archit. Code Optim.*, 8(4):23:1–23:20, January 2012.

[9] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *22nd Annual Net-*

work and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014, 2015.

[10] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 488–501. ACM, 2017.

[11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, Washington, DC, USA, 2011. IEEE Computer Society.

[12] Berk Gülmezoğlu, Mehmet Sinan İnci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. *A Faster and More Realistic Flush+Reload Attack on AES*, pages 111–126. Springer International Publishing, Cham, 2015.

[13] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *ASPLOS'13, 2013*, pages 265–278, New York, NY, USA, 2013. ACM.

[14] Andrew Hoog and Katie Strzempka. *iPhone and iOS forensics: Investigation, analysis and mobile security for Apple iPhone, iPad and iOS devices*. Elsevier, 2011.

[15] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vtz: Virtualizing arm trustzone. In *In Proc. of the 26th USENIX Security Symposium*, 2017.

[16] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes. In *The proceedings of 2015 IEEE Symposium on Security and Privacy*, pages 591–604, San Jose, CA, 17-21, May 2015. IEEE.

[17] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. *Wait a Minute! A fast, Cross-VM Attack on AES*, pages 299–319. Springer International Publishing, Cham, 2014.

[18] ARM Limited. Coremark benchmarking for arm cortex processors. `https://static.docs.arm.com/dai0350/a/DAI0350A_coremark_benchmarking.pdf`. Accessed 2013.

[19] F. Liu and R. B. Lee. Random fill cache architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 203–215, Dec 2014.

[20] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015.

[21] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. *arXiv preprint arXiv:1703.06986*, 2017.

[22] D. Page. Defending against cache based side-channel attacks. *Information Security Technical Report*, 8(1):30–44, April 2003.

[23] D. Page. Partitioned cache architecture as a side-channel defence mechanism, 2005. page@cs.bris.ac.uk 13017 received 22 Aug 2005.

[24] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[25] Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. Ltzvisor: Trustzone is the key. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[26] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[27] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, January 2001.

[28] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[29] Sergei Skorobogatov. Fault attacks on secure chips. *Design and Security of Cryptographic Algorithms and Devices*, 2011.

[30] Fatemeh Tahmasbi, Neda Moghim, and Mojtaba Mahdavi. Adaptive ternary timing covert channel in ieee 802.11. *Security and Communication Networks*, 9(16):3388–3400, 2016.

[31] Randy Torrance and Dick James. The state-of-the-art in semiconductor reverse engineering. In *Proceedings of the 48th Design Automation Conference*, pages 333–338, 2011.

[32] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[33] Kam S Tso, Michael J Pajevski, and Bryan Johnson. Access control of web and java based applications. In *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*, pages 320–325. IEEE, 2011.

[34] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-vm side-channels. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 687–702, San Diego, CA, August 2014. USENIX Association.

[35] Zhenghong Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 83–93, Nov 2008.

[36] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, 2007.

[37] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 9–9, 2012.

[38] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Trans. Netw.*, 23(2):603–614, April 2015.

[39] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, May 2015.

[40] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[41] Ning Zhang, Kun Sun, Wenjing Lou, and Tom Hou. Case: Cache-assisted secure execution on arm processors. In *The 37th IEEE Symposium on Security and Privacy (S&P)*, SAN JOSE, CA, May 23-25 2016. IEEE.

[42] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 313–328, Washington, DC, USA, 2011. IEEE Computer Society.

[43] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on*

Computer and Communications Security, CCS '14, pages 990–1003, New York, NY, USA, 2014. ACM.

[44] Yinqian Zhang and Michael K. Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications security*, CCS '13, pages 827–838, New York, NY, USA, 2013. ACM.

[45] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. arXiv preprint, arXiv:1603.05615v1, 2016. `http://arxiv.org//`.

[46] Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Automatic fingerprinting of vulnerable ble iot devices with static uuids from mobile apps. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1469–1483, 2019.

VITA

Naiwei Liu is a Ph.D. candidate in the University of Texas at San Antonio (UTSA). He joined UTSA in January 2015, and started his studying and research since then. He is advised by Dr. Meng Yu and Dr. Ravi Sandhu, working in UTSA ICS Lab in the year 2018 to 2020.

Naiwei Liu's research focus is on ARM security and system security. At the time in ICS Lab in UTSA, he design and implement security protection framework on ARM platform, and have both theoretical and experimental research related to TrustZone on ARM platform. These research projects lead to some publications, and some working papers.