MSc THESIS

Exploitation of cache based side channels on ARM

Parul Gupta

Abstract

Android smartphones collect and compile a huge amount of sensitive information which is secured using cryptography. There is an unintended leakage of information during the physical implementation of a cryptosystem on a device. Such a leakage is often termed as side channel and is used to break the implementation of cryptographic algorithms. In this work, we utilize cache memory based side channels on android smartphones to retrieve crypto-process information. These side channels are based on the information leakage through the operating system, micro-architecture of the processor and the state of the processor's memory cache. We demonstrate the retrieval of data dependent memory access patterns using a spy application running in the background to recover the full secret key of cryptographic primitives such as AES T-table implementation in OpenSSL, all that would be necessary is a rogue app downloaded from an app store that is run under normal privileges. We show that a mathematical correlation which depends on the guessed key, can be utilized to recover the *complete* key in access-driven cache attacks (CAs). We show the effectiveness of the proposed method using access time measured in noisy environments. We analyze the changes in the correlation values with the number of plaintexts/ciphertexts for a successful attack using key estimation. Furthermore, we discuss and demonstrate the applicability of cache memory based side channel attacks on a white-box implementation of AES.

Faculty of Electrical Engineering, Mathematics and Computer Science

Exploitation of cache based side channels on ARM

Correlation Analysis of Access-driven Cache Attacks on Android Smartphones

---

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

Embedded Systems

by

Parul Gupta
born in Jaipur, India

Cyber Security
Department of Intelligent Systems
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Exploitation of cache based side channels on ARM

by Parul Gupta

Android smartphones collect and compile a huge amount of sensitive information which is secured using cryptography. There is an unintended leakage of information during the physical implementation of a cryptosystem on a device. Such a leakage is often termed as side channel and is used to break the implementation of cryptographic algorithms. In this work, we utilize cache memory based side channels on android smartphones to retrieve crypto-process information. These side channels are based on the information leakage through the operating system, micro-architecture of the processor and the state of the processor's memory cache. We demonstrate the retrieval of data dependent memory access patterns using a spy application running in the background to recover the full secret key of cryptographic primitives such as AES T-table implementation in OpenSSL, all that would be necessary is a rogue app downloaded from an app store that is run under normal privileges. We show that a mathematical correlation which depends on the guessed key, can be utilized to recover the *complete* key in access-driven cache attacks (CAs). We show the effectiveness of the proposed method using access time measured in noisy environments. We analyze the changes in the correlation values with the number of plaintexts/ciphertexts for a successful attack using key estimation. Furthermore, we discuss and demonstrate the applicability of cache memory based side channel attacks on a white-box implementation of AES.

Laboratory : Cyber Security
Committee Members :

Advisor:	Dr. Christian Doerr, Cyber Security, TU Delft
Advisor:	Ruben Muijrers, Security Analyst, Riscure B.V.
Chairperson:	Dr. Jan van der Lubbe, Cyber Security, TU Delft
Member:	Dr. Stephan Wong, Computer Engineering, TU Delft

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

NUMA Non-Uniform Memory Access

ECC Elliptic Curve Cryptogrpahy

MMU Memory Management Unit

RSA Rivest Shamir Adleman

AES Advanced Encryption Standard

PSC Power Side Channels

ACCI Amba Cache Coherent Interconnect

DFA Differential Fault Analysis

DRAM Dynamic Random Access Memory

SCU Snooping Control Unit

MESI Modified Exclusive Shared Invalid

MOESI Modified Owned Exclusive Shared Invalid

ACP Accelerator Coherency Port

ACE Accelerator Coherency Extension

POU Point of Unification

POC Point of Coherence

VIPT Virtually Indexed Physically Tagged

PIPT Physically Indexed Physically Tagged

WBC White-Box Cryptography

Acknowledgements

It has been a period of intense learning for me, not only in the scientific arena, but also on a personal level. I would like to reflect on the people who have supported and helped me so much throughout this period.

I would first like to thank my thesis advisor Dr. Christian Doerr of the Department of Cyber Security at the Delft University of Technology. Dr. Doerr supported me greatly and was always willing to help me. I would then like to thank my colleagues from my internship at Riscure B.V. for their wonderful collaboration. It was indeed a pleasure to work with Mr. Ruben Kuipers from Riscure B.V. Ruben was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right the direction whenever he thought I needed it.

I would also like to thank the experts, Mr. Nikita Abdullin and Ms. Valentina from Riscure B.V. who were involved in the brainstorm sessions for this research project. Without their passionate participation and input, this research work could not have been successfully conducted.

I would like to thank Dr. Jan van der Lubbe for his deep insights and constructive feedback on our research. I would also like to acknowledge Dr. Stephan Wong of the Department of Computer Engineering, and I am gratefully indebted to him for his very valuable comments on this thesis.

Finally, I must express my very profound gratitude to my parents and to my husband for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you everyone.

Parul Gupta
Delft, The Netherlands
August 23, 2017

1 Introduction

With over 1 billion smartphones and 179 billion mobile applications downloaded per year [27], mobile application development is an integral part of the digital ecosystem. Mobile applications are not just redefining marketing strategies but are also revolutionizing the future technologies such as IoT (Internet of Things). As per a report by Gartner [68], there will be 21 billion connected devices by 2020 which includes smart objects like LED light bulbs, toys, domestic appliances, sports equipment etc. As IoT will continue to evolve at an ever-increasing pace, smartphones will function as the main interface through which we will be able to interact with IoT enabled devices. Smartphones will act as remote controls, displaying and analyzing information, interfacing with social networks, paying for subscription services, updating object firmware etc. With 87% market share [15], android clearly dominates the smartphone market and it is predicted that in 2017 around 1 billion android smartphones will be shipped. One of the reasons behind such a huge success is that android is considered to be "open and free". Also for developers, it allows them to incorporate already available third party code in their applications which can be confirmed by the heavy usage of native libraries among the most popular applications. However, including third party code in an application can have some severe consequences both on the user and the developer of the app.

Mostly, users download android applications from the Play Store and review the permission requirements presented to them and then install the applications. In addition to the Play Store, users can make use of other application stores and there exists multiple ways to install new applications onto an android smartphone. However, post manufacturer software downloads from 3^{rd} party market places or application stores provide opportunities for malicious mobile applications to gain access to the devices. Such applications can have or request for an access to personal information (including saved passwords, login information and email-access) with or without user's knowledge. Hence, apart from empowering mobile devices these applications can reveal a great deal of information about its users. Prior work has shown that many third-party applications request more permissions than needed. Felt et al. [34] found out that one third of 940 apps on the android Play Store request access to the information without any legitimate need. It makes smartphones attractive targets and represents a significant risk to information security and data security of the user.

Interestingly, android smartphones are susceptible to leakage of user's information in unexpected ways as explained in [58][53][35][19][48][52]. Android(Figure: 1.1)[3] uses a kind of Unix Sandboxing method to run the applications and android software stack is shown in Figure: 1.1[3]. Usually mobile applications run with different permissions and privileges and low-level implementation of machine in collaboration with OS provides

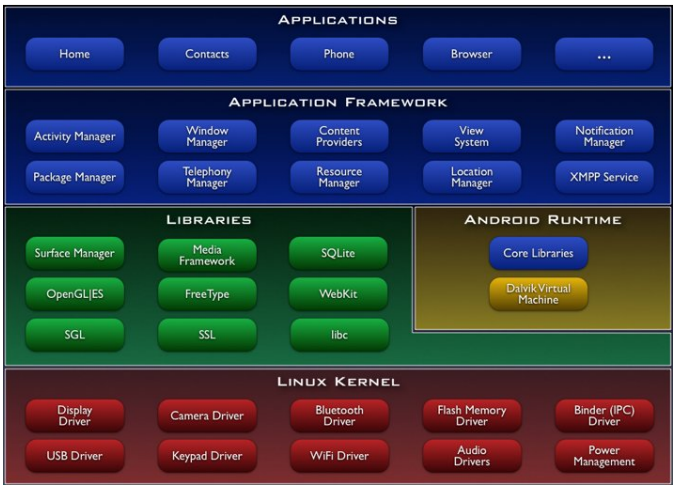
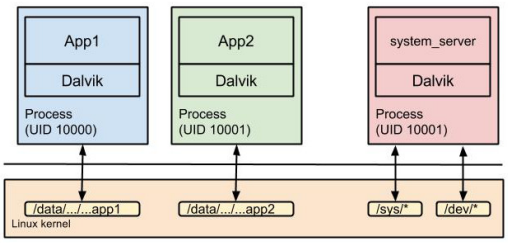
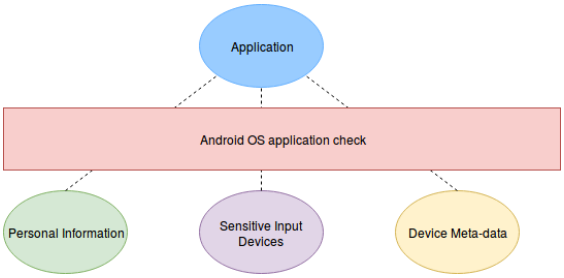


Figure 1.1: Android Software Stack

for desired access control. For example, each application that is installed on an android device is assigned its own unique user identifier (UID) and group identifier (GID). This behavior is different than conventional Linux, where applications that are shared by a user are run under user's context. It also relies on the fact that user is not installing any malicious software on his/her computer because there is no protection mechanism against accessing files that are owned by the same user that you are running as. Amongst various other strategies are kernel and userspace separation, filesystem permissions etc.



(a) Android Sandboxing



(b) Android Permission Model for Sensitive Information

As each application has its own user name and memory space, one application cannot access the resources of other application until defined explicitely (Figure: 1.3). Each application runs into separate VMs (Figure: 1.2a). It implies that vulnerability found in one application will not affect remaining applications. There exists a permission enforcement by Android in order to protect the following:

- Access to sensitive APIs
- Access to content providers

build

- Inter and Intra-application communication

The permission model (Figure: 1.2b)[5] is all-or-nothing; the user can install the app or not, but cannot choose to install it with reduced permissions. This imparts a significant responsibility to both the developer (to accurately specify required permissions) and the user (to understand the risk invloved and make an informed decision). Additionally, it is a developer's duty to protect user's device from information security breaches by following safe coding practices.

In practice, cryptographic primitives and cryptographic protocols are implemented to protect user's information from malicious applications. Smartphones use cryptographic keys to protect sensitive information such as health records and banking passwords. Several techniques such as disk encryption and the encrypted communication over the Internet are applied. These techniques intrinsically make use of standardized cryptographic algorithms such as Advanced Encryption Standard (AES) etc. In cryptographic implementations, the secret key directly affects the emitted side-channel information for e.g. power usage patterns, observations made on this leaked data can eventually lead to the revelation of the secret key. Hence, such malicious leakage to untrusted third-party applications which exploit these side-channels is one of the key challenges to the mobile data security and privacy.

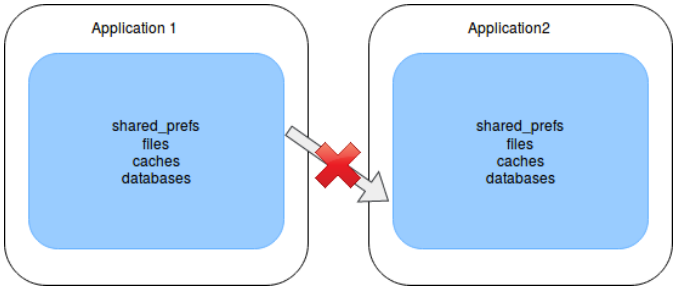


Figure 1.3: Applications on Android

A potential leakage in the form of execution footprints [19] leaks timing information such as, access time to perform a look-up in a table while performing a crypto operation. Such a side-channel originally stems from the microarchitectural structure of the underlying microprocessor. In the same direction, Osvik et.al [58] show how a low-level implementation detail of modern CPUs, namely the structure of memory caches, leads to cross-process information leakage between processes running on the same processor. In essence, the cache forms a shared resource which all processes compete for, and it thus affects and is affected by every process. While the data stored in the cache is protected by virtual memory mechanisms, the metadata about the content of the cache, and hence the memory access patterns of processes using that cache, is not fully protected [58][19][53].

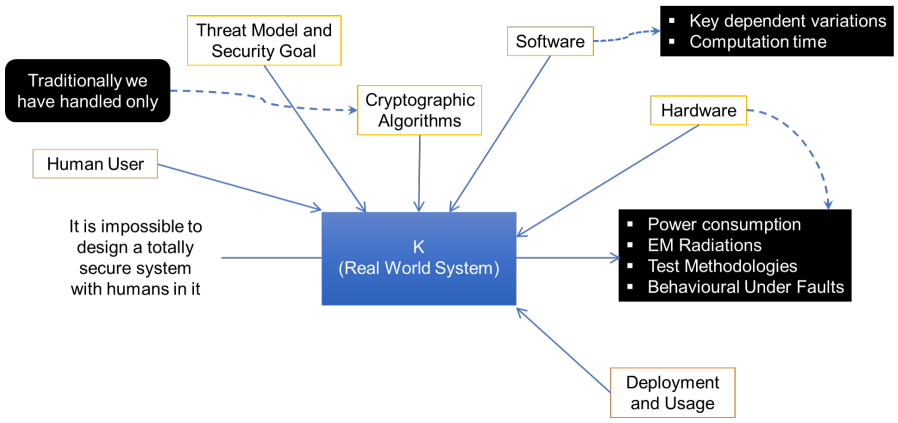


Figure 1.4: Side Channels Overview

Several solutions are used to protect the cryptographic keys which range from unprotected software implementations to temper resistant hardware implementations. One such implementation is a white-box implementation [74] which hides the cryptographic key inside a software and turns a keyed cryptographic algorithm into an unintelligible program with the same functionality. Thus, a white-box secure program can then be executed in an untrusted environment without the fear of exposing the underlying keys. The code itself is tamper-proof and robust, just as a secure element. White-box implementations [33][24] can be used to protect smartphones against malicious applications. Other use-cases include the protection of digital assets (including media, software and devices) in the setting of digital rights management, the protection of Host Card Emulation (HCE) and the protection of credentials for authentication to the cloud. In a typical white-box attack set-up, it is assumed that the adversary has full control over the execution environment which is more realistic in comparison to black-box or gray-box attack model. Additionally, if one has access to a "perfect" white-box implementation of a cryptographic algorithm, then this implies one should not be able to deduce any information about the secret key material used by inspecting the internals of this implementation. Ideally, a white-box implementation [33] should resist all existing and future side-channel attacks. However, in our research we demonstrate using side-channel attacks that white-box implementations are vulnerable to cache attacks.

1.1 Related Work

In 1999, the concepts of Simple Power Analysis (SPA) and Differential Power Analysis (DPA) were introduced by Kocher et al.[48][49], where an attacker can extract cryptographic keys by studying the power consumption of a device. They explained that there is a leakage of information from computers and chipsets about the operation they execute. They utilized the power consumption measurements to find secret keys from tamper resistant devices.

A different side channel on modern computing architectures is introduced by the memory hierarchy that stores subsets of the computer's memory in smaller but faster memory units, so-called caches. Cache side-channel attacks exploit the different access times of memory addresses that are either held in the cache or the main memory. Kelsey and Kocher et al. [46][48] were the first to discuss the theore tical cache attacks. Page and Tsunnoo et al. [59][67] discussed the applicability of cache attacks on DES. Bernstein [20] demonstrated complete AES key recovery from known-plaintext timings of a network server. It has been one of the most extensively investigated attacks. Hund et al. [44] demonstrated how an adversary can implement a generic side channel attack against the memory management system to deduce information about the privileged address space layout. Gullasch et al. [41] attacked the L1 cache and demonstrated the exploitation of shared memory to mount cache attacks. The Evict+Time and Prime+Probe techniques by Osvik et al. [58] explicitly targeted cryptographic algorithms. While Herath et al. [13] discussed the CPU Hardware Performance Counters for Security, Chiappetta et al. [28] demonstrated the real time detection of cache-based side channel attacks using hardware performance counters. Yarom and Falkner introduced Flush+Reload attack in 2014 where the target was L3 cache instead of L1 cache. It allows an attacker to determine which specific parts of a shared library or a binary executable have been accessed by the victim with an unprecedented high accuracy. Based on this work Gruss et al. [53] demonstrated the possibility to exploit cache-based side channels via cache template attacks in an automated way and showed that besides efficiently attacking cryptographic implementations, it can be used to infer keystroke information and even log specific keys. Zhao et al. [76] presented an access-driven attack on the first and second round of the AES encryption. Laradoux et al. [51] explained about the collision attacks on processors with cache and countermeasures.

However, it is also possible to induce hardware faults by software, and thus from a remote location, if the device could be brought outside of the specified working conditions. In 2014 Kim et al. [47] demonstrated that accessing specific memory locations at a high repetition rate can cause random bit flips in Dynamic Random-Access Memory (DRAM) chips. Since DRAM technology scales down to smaller dimensions, it is much more difficult to prevent single cells from electrically interacting with each other. They observe that activating the same row in the memory corrupts data in nearby rows. Gruss et al. [53][39] showed that such bit flips can also be triggered by JavaScript code loaded on a website. However, this attack can only be demonstrated on Intel and AMD systems using DDR3 and modern DDR4 modules. Brumley et al. [26][25] carried out the attack on live cache-timing data and cache storage data. In 2015, Seaborn demonstrated that cache side channel attack could be exploited for privilege escalation. Weiss et al. [71] and Bogdanov et al. [22] attacked ARM7 microcontrollers and ARM Cortex-A8 processors. Van der Veen et al. [36][69] investigated the rowhammer bug [47] on ARM-based devices as well. Key rank estimation techniques came as a major breakthrough in the field of estimation of security of a cryptographic implementation. Vincent et al. [61] explained a simple key enumeration and rank estimation technique using Histograms.

1.2 Motivation

In the current scenario [8], most of the side-channel attacks on a smartphone require an adversary to have physical access to it or at least have a cable or probe in close proximity to the device while it is performing some cryptographic operation. Usually the time to attack an application is dependent on the application specification for e.g. time to hack RSA [62] would be different than that required for ECC [42]. Such impractical requirements usually result into futile attempts while attacking a mobile device remotely. However, such unintended information leakage poses greater threats than once perceived. Unintended data leakage [14] occurs when a developer accidentally places sensitive information or data in a location on the mobile device that is easily accessible by other apps on the device. Such unintended data leakage or side-channel data leakage [9] from mobile devices arises from the vulnerabilities in the following :

- Operating System
- Frameworks
- Compiler environment
- Hardware
- Microarchitecture of CPU

For instance, a developer's code processes sensitive information supplied by the user or the backend. During that processing, a side-effect (that is unknown to the developer) results in that information being placed into an insecure location such as cache etc. on the mobile device that other apps on the device may have open access to. Under these assumptions, we can safely assume that one malicious app can extract valuable information about other apps running on the same device. Typically, cache side-effects originate from the underlying mobile device's operating system (OS). It is fairly easy for a rogue application to detect data leakage by inspecting all mobile device locations that are accessible to victim app for the app's sensitive information. Several remote attack vectors which can be potentially exploited on an Android OS for such attacks are :

- Browsers and Document Readers
- Custom Update Mechanisms
- Remote Loading of Code
- WebViews
- Listening Services
- Messaging Applications

Interestingly, if the appropriate security applications and procedures are not applied, these rogue applications can utilize side-channel leakage to extract the secret key used to encrypt user's private information. For example, a spy application may determine the secret key information by using the memory access pattern of a cryptographic algorithm running in the backend of a banking application.

1.3 Challenges and Results

The prime focus of our research is on the cache based side channels on android smartphones. Cache memory is a smaller and faster storage area in comparison to the main memory and therefore many different addresses in main memory are mapped to the same cache entry. The cache architectures are optimized to minimize the number of cache misses for typical access patterns but can be easily manipulated. Depending on the cache replacement policy, such property can be used to perform manual cache eviction and monitor the cache behavior. Additionally, a data item residing in a cache (cache hit) is retrieved much faster than a data item that is not in the cache (cache miss) and the difference in accesses is measurable. Such properties of the cache can be exploited to retrieve the memory access patterns during the encryption/decryption process of a cryptosystem.

We studied the cache behaviour using the side-channel leakage on android smartphones. Existing cache-based side channel attacks have thrived on x86 architecture which fulfills almost all the necessary requirements. For instance, the x86 architecture provides with *clflush* instruction to evict a cache line. An accurate timing mechanism which employs *rdtsc* instruction is also available in unprivileged mode. However, android smartphones use multi-core ARM CPUs which have different architecture and different instruction set than Intel CPUs. In contrast to x86, there are several challenges on ARMv7 CPUs which can be enumerated as follows :

1. Non-inclusive/exclusive cache
2. No unprivileged cache flush instruction
3. Pseudo random cache replacement policy
4. Accurate unprivileged timing source

Gruss et.al [53] developed a library *libflush* in order to mount cache-based side channel attacks on x86 and ARM processors. To our advantage, we have used *libflush* to perform cache attacks on ARMv7 based processors such as Krait 400 (similar to Cortex A-15). Using libflush [53], not only could we overcome all the above mentioned challenges but could also attack white-box cryptosystems. We start with exploring how cache attacks (using shared memory) can be used by a rogue application to spy on a victim application performing look-ups. We refer to the work of Osvik et.al [58] in reducing the entropy of OpenSSL AES implementation from 128 bits to 64 bits on Nexus 5 by attacking first round of AES. We develop a novel strategy using techniques mentioned in [58] and [39] and successfully recover the full key by incorporating *correlation analysis* of cache traces on x-86 and ARM. We also discuss the applicability of *key rank estimation* techniques on the generated cache traces. We analysed our results with resepct to various attack parameters. Several statistical properties of cache traces have also been discussed. Eventually, we explore the applicability of cache attacks on a white-box implementation of AES on Nexus 5.

1.4 Research Goal

In the recent 10-15 years [79], there has been a shift in attackers' approach while attacking a cryptosystem which was previously based on exploiting the mathematical weakness of cryptographic algorithm itself. Now adversaries try to find a correlation between the side channel information and the operation related to the secret key. Side channel attacks focus on the way cryptographic algorithms are implemented rather than the algorithm itself. These attacks work because there is a correlation between the physical measurements (e.g., computation time, power consumption etc.) taken during computations and the internal state of the processing device, which is itself related to the secret key.

Our research agenda is to deploy cache based side channel attacks (later referred as "cache attacks or CAs") on android based smartphones. We identify the leakage associated with the hidden secret key in gray-box and white-box cryptosystems. With this work, we aim to find answer to the following research question:

How to successfully recover the full key from crypto applications on android smartphones using cache based side channels along with statistical analysis?

Eventually, we find answers to the following sub-questions with respect to our test devices (1.6):

- How can we use cache attacks to perform cryptanalysis of a cryptosystem relying on information related to cache hit or miss?
- How can we make use of correlation analysis on cache traces to extract a secret key of a cryptographic algorithm?
- How statistical evaluation techniques like key rank estimation assist in reducing the key entropy while extracting the secret key of a cryptographic algorithm ?
- Why does the initial cache state influences the performance of cache attack?
- How do different attack scenarios and parameters create an impact on the overall intensity of the attack?
- How can we use cache attacks to perform cryptanalysis of a white-box cryptosystem implementation on an android smartphone?

1.5 Outline

In this research work, we discuss the cache attacks on the execution of artifical and crypto applications on android smartphones. We demonstrate the cache attacks using a spy process which monitors the execution of crypto process/victim process on an android smartphone. In Chapter 1, we briefly discuss the issues associated with third party native code on android and how android fails to safeguard user's sensitive information

against cache based side-channels. We introduce our research goal alongwith the motivation behind it. In Chapter 2, we describe the basic concepts about cache organisation, functioning and maintainence. Also, we discuss the OS vulnerabilities and how they were utilised in the past to attack applications on x86 architecture. In Chapter 3, we introduce side channels on android smartphones. We categorically discuss cache attacks on ARM. We discuss the portability of cache attacks mountable on x86 to ARM architecture. A brief introduction to the OpenSSL implementation of AES is complemented with the discussion about Correlation Analysis. In Chapter 4, we describe the adversary attack model, attack assumptions and attack strategy. Finally, we highlight attack results (PRIME+PROBE and EVICT+RELOAD attack) and related conclusions. The refinement of our resusing Key estimation is also shown. Chapter 5 discusses the results and in chapter 6, we highlight some of the major contributions to the field of cache attacks which are also a major source of inspiration for our work. Chapter 7, concludes our research and discusses contributions from our end to the field of cache attacks on ARMv7 architecture.

1.6 Test Devices

We aim to perform side channel attacks on Nexus 5 (Android 5.0) which has QUALCOMM KRAIT 400 (approx 2.25 GHZ) processor with following properties of interest:

- Harvard Architecture: 32 bit
- Quad-core
- Local L1 caches (Instruction and Data) per core
- High performance and extremely efficient
- Uses ARMv7-A ISA
- Snoop control unit for cache data coherence

Cache Organization (per core) of KRAIT 400 is as follows:

Level of cache	Cache Size	Associativity	Cache Line Size	Inclusiveness
L1	2x16KB (per core)	4-way	64 Bytes	Non-inclusive and exclu
L2	0.5MB (per core)	8-way	64 Bytes	Shared & unified

Table 1.1: Cache organization Krait 400

Another Test device is Samsung Galaxy S4 (Android 5.0) with following specifications:

- CPU Clock Speed: Up to 1.9 GHz
- Quad-core CPU: 4x Qualcomm Krait 300 CPU
- CPU Bit Architecture: 32-bit

- Snapdragon 600 SoC with Krait 300^{TM}
- ARMv7-A ISA

Cache Organization of KRAIT 300 is as follows:

Level of cache	Cache Size	Associativity	Cache Line Size	Inclusiveness
L1	2x16KB (per core)	4-way	64 Bytes	Non-inclusive
L2	0.5MB (per core)	8-way	128 Bytes	Shared & unified

Table 1.2: Cache organization Krait 300

1.7 About libflush

Gruss et.al [53] developed *libflush* to deploy cache attacks on x86 as well as ARMv7 and ARMv8 architectures. It allows us to easily implement attacks based on Prime+Probe, Flush+Reload (another variant is Evict+Reload), Flush+Flush and Prefetch attack techniques. We utilize this library to perform cache attacks on the cryptographic primitives like AES in native code. Also, we use it to determine the leakage from shared libraries used in Android Applications. As an extension to our work, by using *libflush* we aim to mount cache attack on a white-box implementation of AES.

Background 2

2.1 Memory Hierarchy

Memory organization of a processor separates each memory level on the basis of its response time, location, capacity, units of transfer, performance and physical characteristics such as decay, volatility, power consumption. As shown in (Table: 2.1), when performance is a major concern, then cache memories are undoubtedly the most important part of the memory organization.

A cache memory is a small, volatile and fast array of memory placed between the processor and main memory. It is the fastest memory in a computer, and is typically integrated onto the motherboard and directly embedded in the processor or main random access memory (RAM). It is equipped with additional features to cater to high throughput requirements of a processor. It is a holding buffer that stores portions of recently referenced system memory. The processor uses cache memory in preference to system memory whenever possible to increase average system performance. A write buffer is a very small

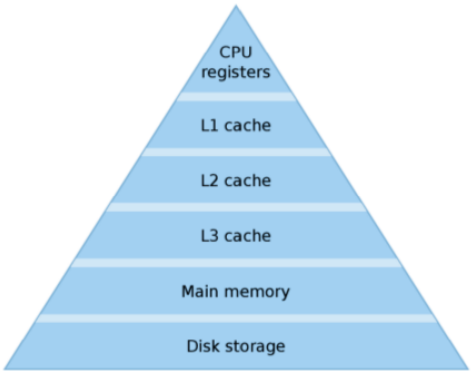


Figure 2.1: Memory Hierarchy

Memory Type	Access Times (in ns)
Registers	1-2
L1 Cache	3-10
L2 Cache	25-50
Memory	30-90

Table 2.1: Access Times

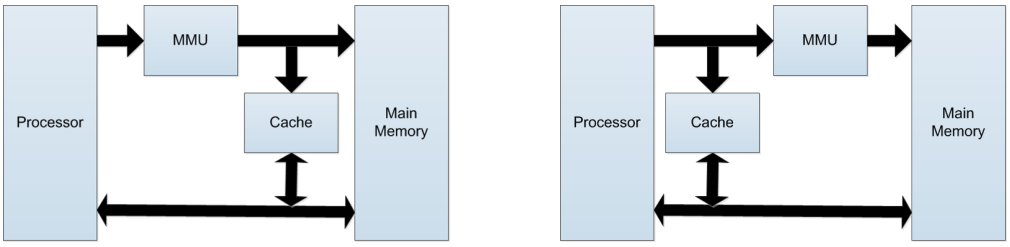
FIFO memory placed between the processor core and main memory which helps free the processor core and cache memory from the slow write time associated with the writing to main memory. The principle of locality of reference states that computer software programs frequently run small loops of code that repeatedly operate on local sections of data memory and explains why the average system performance increases significantly when using a cached processor core. However, the disadvantage of cache memories is that they lead to side-channel information leakage leading to attacks on ciphers. Data cache memory can be a source of leakage for any cipher implementation that uses look-up tables which are accessed at key dependent locations.

2.2 About CPU Caches

As discussed previously, cache [73] provides faster data storage and access by storing instances of programs and data routinely accessed by the processor. Thus, when a processor requests data that already has an instance in the cache memory, it does not need to go to the main memory or the hard disk to fetch the data. When a byte of data must be paged in during the computation, the processor first looks for it in the cache. If present in the cache, this results in a cache hit. The data is brought to the registers within a single clock cycle without stalling the pipeline. If not present in the cache, this results in the cache miss, and the desired data is fetched from Non-Volatile Memory (NVM), and the entire line containing the desired data is loaded into the cache. In the case of a cache miss, the data has to be brought from the memory and it is an expensive operation as shown in Table:2.1.

All the cache design attributes aim towards maximizing the cache hits and improving the overall system performance. Various attributes of cache design which decide the overall access time, transfer rate and performance are as follows:

- **Physical and Logical Caches:** The placement of a cache before or after the Memory Management Unit (MMU) is either physical or logical. A logical cache (Figure: 2.2a) is placed between the processor core and MMU references code and data in a virtual address space. A physical cache (Figure: 2.2b)is placed between MMU and main memory. The references to code and data memory are done using physical addresses.



(a) Logical cache (b) Physical cache

Figure 2.2: Placement of Cache Memory

- **Cache Address Translation:** On the basis of physical and virtual addresses being represented by tag or index (Figure: 2.3), caches can be divided into the following classes :

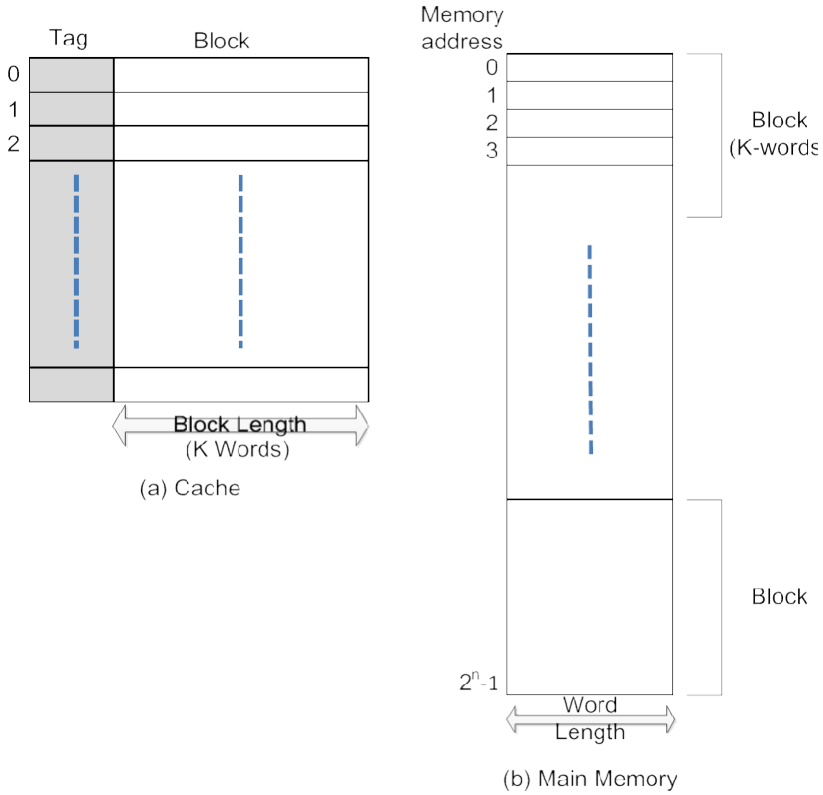


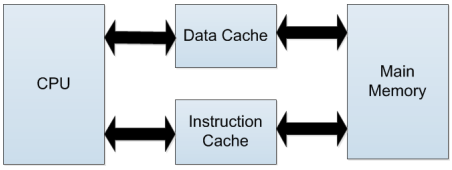
Figure 2.3: Cache Memory and Main Memory Addresses

  - **Physically indexed, physically tagged (PIPT):** These caches [73] use the physical address for both index and tag.
  - **Physically indexed, virtually tagged (PIVT):** Not many architectures implement this cache design [73].
  - **Virtually indexed, physically tagged (VIPT):** These caches [73] use the virtual address for the index and the physical address in the tag **Krait 400 CPU, on our test device (Nexus 5) uses VIPT caches.**

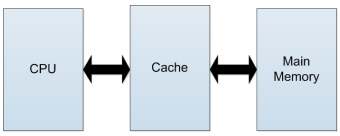
– **Virtually indexed, virtually tagged (VIVT):** These caches [73] use virtual address for both tag and index.

- **Cache Size:** Cache size [73] is an important attribute as it affects the performance. The increase in the size of the cache would result in faster data accesses (upto a certain point) but would cost more as well. It should be noted that checking for data in large caches takes more time.

- **Core bus architecture:** The core bus architecture [73] helps determine the design of a cached system. A Von Neumann (unified) architecture (Figure: 2.4b and 2.4a) uses a unified cache to store instructions and data. Harvard architecture, on the other hand uses a split cache: one cache for instructions named as Instruction Cache and another cache for data named as Data Cache. ARM makes use of (modified) Harvard architecture, it has one L1 instruction cache and one L1 data cache.



(a) Harvard Architecture



(b) Von Neumann(Unified) Architecture

Figure 2.4: Core Bus Architecture

- **Mapping Function:** One key decision in cache design is that how the main memory blocks should be mapped to the cache slots (Figure: 2.3). There are three possible approaches to map memory blocks to cache slots (*we will use slots or lines interchangeably throughout this document*):

  – **Direct mapped cache:** In a direct mapped cache [73] each memory block is mapped to one specific slot in the cache. Hence, it is not possible to store two memory blocks which map to the same cache set simultaneously. In the following steps, we will discuss how address mapping is done.

  Block Identification: Let us assume that the main memory contains n blocks (which requires $log_2(n)$) and cache contains m slots, so n/m different blocks of memory can be mapped (at different times) to a cache slot. Each cache slot has a tag saying which block of memory is currently present in it, each cache slot also contains a valid bit to ensure whether a memory block is in the cache slot currently. Figure: 2.5 demonstrates direct mapped cache functionality. It can be further simplified as follows:
  Number of bits in the tag: $log_2(n/m)$
  Number of sets in the Cache: m
  Number of bits to identify the correct slot: $log_2(m)$

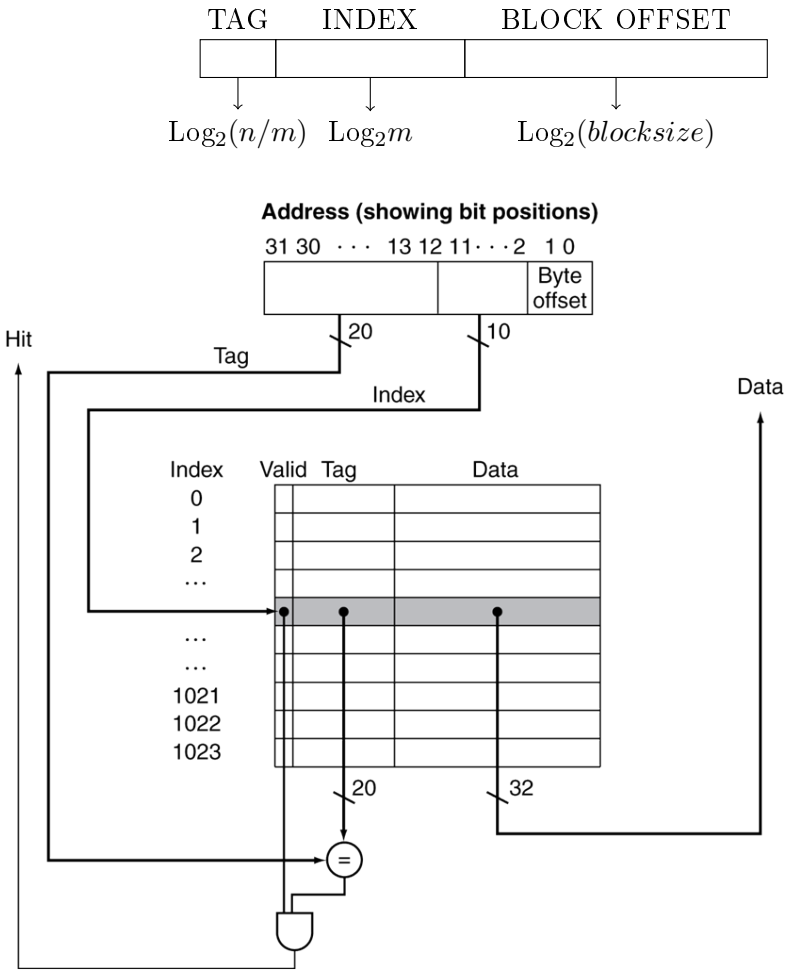


Figure 2.5: Direct Mapped Cache

This mapping strategy (Figure: 2.5) suffers from *poor cache utilization* and *cache thrashing*.

– **Fully associative cache:** Instead of mapping memory blocks to specific cache slots or cache lines, they can be mapped to any cache slot using fully associative caches. Each memory block can be stored in any slot in the cache, effectively like a hash table. Fully Associative cache(Figure: 2.6 and 2.7) has high efficiency as the data can be stored in any entry, but it is expensive in terms of circuit complexity. It requires independent simultaneous ways of access and a comparator for each cache entry. Therefore, the size of this type of cache is very small and used only for specific cases (e.g. TLB). Let the main memory address be divided into two groups which are tags and word bits. Words are low-order bits and identifies the location of a word within a block and tags are high-order bits which identifies the block.
– **N-way set associative cache:** A combination of functionality of associative and direct mapped caches is used in N-way set associative caches [73]. In an

Figure 2.6: Memory address for Fully Associative Cache

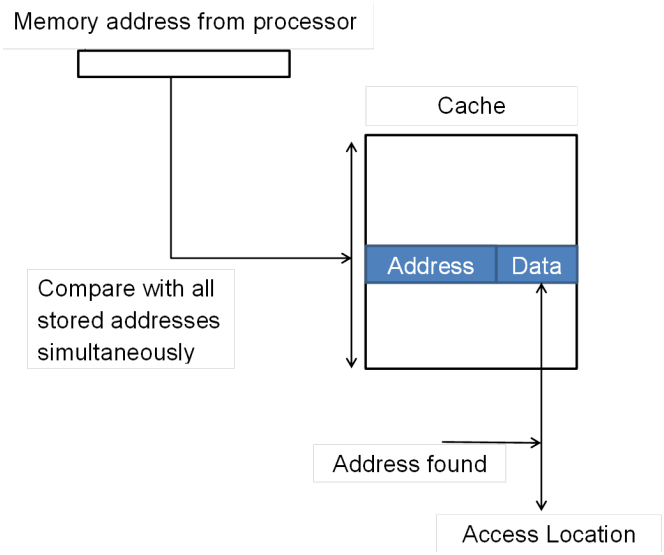


Figure 2.7: Fully Associative Cache

N-way associative cache [73], each memory block can be stored in any one of the N particular slots in the cache. Modern processors use one or more levels of set-associative memory cache.

Such a cache consists of storage cells called cache lines, each consisting of B bytes. The cache is organized into S cache sets, each containing W cache lines, so the cache size is $B.S.W$ bytes. The mapping of memory addresses into the cache is limited as follows. First, the cache holds copies of aligned blocks of B bytes in main memory (i.e. blocks whose starting address is 0 modulo B), which we will term as memory blocks. When a cache miss occurs, a full memory block is copied into one of the cache lines, replacing (*evicting*) its previous contents. Second, each memory block may be cached only in a specific cache set; specifically, the memory block starting at address a can be cached only in the W cache lines belonging to cache set $[a/B \ mod \ S]$. For example, in a 8-way 64 KB ($2^b = 64kB$) cache with a cache line size 64 bytes $2^\delta = 64$, number of cache sets would be $(2^b/2^\delta)$. Each memory block can be stored in 8 different cache slots in a cache set. Commonly, blocks with indices with the same lowest order bits will all compete for 8 slots. Mapping scheme will be based on below mentioned equations in an n-way set-associative cache.

build

$$A_{word} = A mod\, 2^{\delta}$$
$$A_{set} = \lfloor A/2^{\delta} \rfloor\, mod 2^{s}$$
$$A_{tag} = \lfloor \lfloor A/2^{\delta} \rfloor /2^{s} \rfloor$$

- **Cache Replacement Algorithm:** These optimizing algorithms [73] decide how to make room for the new entries in the cache. It involves operations like cache clean, cache flush or cache eviction. They are described as follows :
  - **Cache Clean:** A cache clean [73] operation causes the data present in the cache line to be written back to the next level of cache or to the the memory if the cache line is tagged as "dirty". In such a case, cache line holds the latest copy of the data.
  - **Cache Invalidate:** This function invalidates a cache line so that future reads go the main memory instead of that cache line. The invalidated lines don't get written back to the memory. Hence, there is a chance that we may loose data. It is suggested to perform a clean operation in prior to invalidation.
  - **Cache Evict:** In the case of eviction [73], a cache line is written back to the memory when the cache is full. Cache eviction can be accidental or forced and the consequences of eviction depend on our cache configuration. This would be further explained in the later sections.
  - **Cache Flush:** It is similar to cache eviction as a cache line or the entire cache is flushed to the main memory. However, the purpose differs, for e.g. when a device reads memory contents, cpu flushes the cache lines to the memory before the read happens so that the device, reading memory contents, gets the most updated data.

  Various types of cache replacement strategies are described as follows:
  - **Least Recently Used (LRU):** It replaces least recently used cache slots. It is used in set associative caches.
  - **Least Frequently Used (LFU):** It replaces cache slots with minimum number of cache hits. It is most efficient algorithm but highly expensive and hence commonly not used.
  - **First-In First-Out (FIFO):** It replaces slots which have been longest in the cache. It is used in set associative caches.
  - **Round Robin:** It is used in full associative caches.
  - **Random Replacement (RR):** It does not make use of access history and randomly selects a candidate for replacement. It is used in full associative caches
  - **Pseudo-Random LRU (PLRU):** It almost always discards some of the least recently used items in the cache. It is used in set associative caches. In the later sections we will discuss how it is used in ARM architectures.
- **Write Policy:** The cache write policy determines how it handles writes to memory locations that are currently being held in cache. It means that the

cache blocks should not be overwritten until and unless main memory is up to date.

Most common write policies are :

- **Write-through:** Under this policy, data is concurrently written both to cache and to main memory, or first to cache and then to memory. In such a case multiple CPUs can monitor main memory traffic to keep local (to CPU) cache up to date. It generates lots of traffic and slows down writes.
- **Write-back:** Write-back policy updates data only in the cache. The updated data is "written back" to the main memory when needed, for instance on cache line replacement (overwrite) or when required by other caches. This reduces bus and memory traffic because the next cache line update is taken only in the cache without involving the memory.

- **Block Size:** Cache memory retrieves [73] not only the desired word but also the adjacent words, hence on increasing the cache line size we will experience an increase in cache hits up to a certain extent. This results in performance enhancement based on the principal of locality. However increasing block size beyond a certain point can also cost performance penalty because it will reduce the number of blocks which can fit in the cache. Also, the data will be overwritten at a much faster pace after being fetched. Not only this, each additional word will be less local and hence will be less used. Cumulatively, the probability of using newly fetched information will become less than the probability of reusing replaced.

- **Cache Coherency:** In systems such as Symmetric Multiprocessor System [72][73], multi-core systems etc. where a dedicated cache for each processor, core or node is used, a consistency problem may occur when the same data is stored in more than one cache. This problem arises when a data is modified in one cache. This problem can be solved in two ways:

  1. Invalidate all the copies on other caches (broadcast-invalidate)
  2. Update all the copies on other caches (write-broadcasting), while the memory may be updated (write through) or not updated (write-back).

  Usually coherency is more of an issue for data cache then instruction cache. Three approaches are adopted to maintain the coherency of data.

  1. **Bus watching or Snooping:** It is generally used for bus-based Symmetric Multiprocessor System (SMP), multi-core, Non-Uniform Memory Access (NUMA) systems. In ARM bus snooping is used to maintain cache coherence.
  2. **Directory-based Message-passing:** It may be used in all systems but typically in NUMA system and in large multi-core systems. In this approach the sharing status of a particular cache line is kept in one location called directory.
  3. **Shared cache:** It is generally used in multi-core systems.

- **Number of Caches:** Multi-level cache [73] is a trade-off between price and performance of modern CPUs. It provides an efficient way to coordinate multi-core

processors. Cache memory is made up of SRAMs which will retain its value indefinitely as long as it has operating power, making it much faster. SRAMs come at multiple speeds and have price-band depending on their operating speeds. Hence in the cache hierarchy, each cache has a different size, speed and corresponding cost. In a multi-core processor architecture, each core has its own L1 cache, writing to and reading from L1 cache does not suffer interference from other cores.

Usually L1 cache is faster and smaller than L2 or L3 and hence, it is more expensive. Therefore, the processor will put the values it needs the most in the faster cache (L1) and the information that it needs less in a slower cache (L2 or higher). In multi-level cache organization caches can be exclusive, inclusive and non-inclusive on the basis of the manufacturer's choice. In exclusive caches, last level cache (LLC) does not hold the copies of data present in lower-level caches. In inclusive cache design, the LLC holds the copy of data present in lower-level caches. But, a non-inclusive cache is neither inclusive nor exclusive. The cacheable properties 2.8 of normal memory are specified separately as inner and outer attributes. Typically, inner attributes are used by the integrated caches, and outer attributes are made available on the processor memory bus for use by external caches.

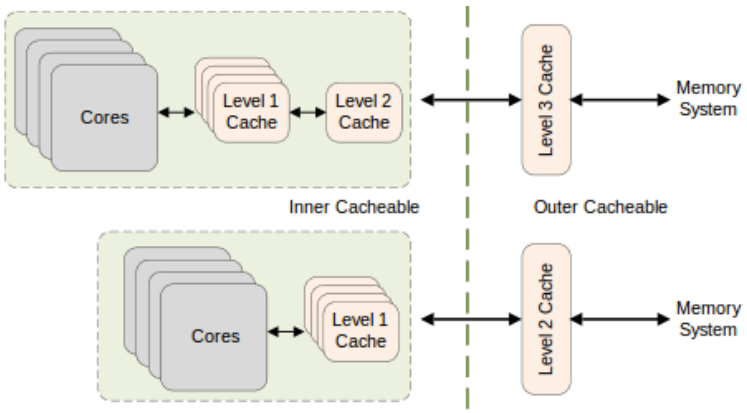


Figure 2.8: Multiple Cache Organization

2.2.1 Summary

There are several important aspects of memory hierarchy which are worthmentioning and can be summarized as follows:

1. Memory accesses are not performed in constant time.

2. Cache memory is fastest amongst all (except registers) in the memory hierarchy and is an integral part of modern processor architecture.

3. A CPU cache hides the latency of main memory by keeping copies of frequently used data

4. Cache hits and misses are distinguishable even on noisy systems on the basis of clock cycles they consume.

5. Last Level Cache (LLC) is usually a shared resource. It can be inclusive, exclusive or non-inclusive based on the implementation of processor architecture.

6. Cache coherency is important in order to maintain uniformity in shared data resource in multi-core processors or multi-processing systems.

7. The cache coherence on multi-processor systems ensures that the data is found independently of where in the cache it is stored.

8. Data cached anywhere in the on a multiprocessor system has lower access times than the memory accesses facilitated by fast interconnects.

2.3 ARM Caches

2.3.1 Cache Organization

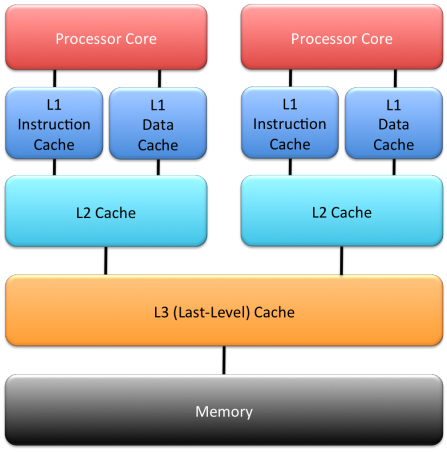


Figure 2.9: ARM Cache Architecture

ARM architectures [17] use set-associative cache design (Figure: 2.10) and (Figure: 2.9)[43] where the degree of associativity and the line size varies. It also makes use of First-In-First-Out(FIFO) write buffer to enhance memory write performance. The write buffer is interposed between the cache and the main memory and consists of a set of addresses and a set of data words.

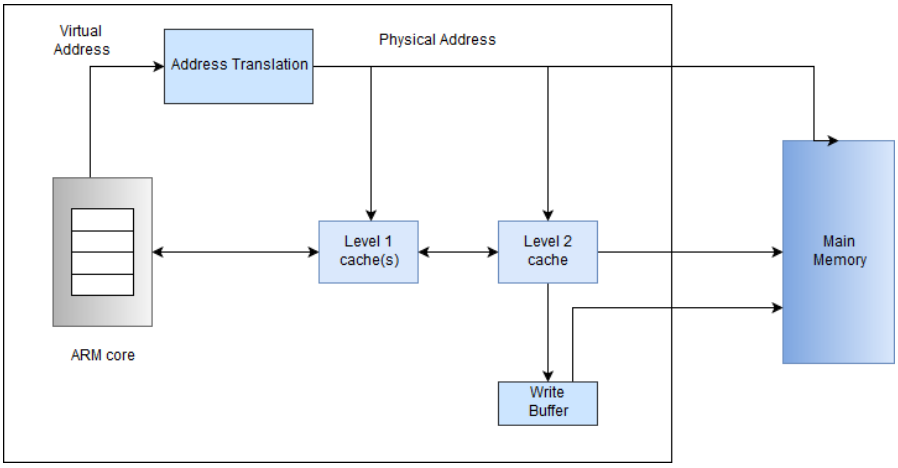


Figure 2.10: ARM Cache Organization

2.3.2 Cache Coherency

The Snooping control unit (SCU) [6] uses hybrid Modified Exclusive Shared Invalid (MESI) and Modified Owned Exclusive Shared Invalid (MOESI) protocols to maintain coherency between the individual L1 data caches and the L2 cache. The L1 data caches support the MESI protocol. The L2 memory system contains a snoop tag array that is a duplicate copy of each of the L1 data cache directories. The snoop tag array reduces the amount of snoop traffic between the L2 memory system and the L1 memory system. Any line that resides in the snoop tag array in the Modified/Exclusive state belongs to the L1 memory system. Any access that hits a line in this state must be serviced by the L1 memory system and passed to the L2 memory system. If the line is invalid or in the shared state in the snoop tag array, then the L2 cache can supply the data.

The SCU contains buffers that can handle direct cache-to-cache transfers between cores without reading or writing any data on the ACE. Lines can migrate back and forth without any change to the MOESI state of the line in the L2 cache.

Shareable transactions on the ACP are also coherent, so the snoop tag arrays are queried as a result of ACP transactions. For reads where the shareable line resides in one of the L1 data caches in the Modified/Exclusive state, the line is transferred from the L1 memory system to the L2 memory system and passed back on the ACP.

2.3.3 Cache Policies

The cache policies [7] enable the user to describe when a line should be allocated to the data cache and what should happen when a store instruction is executed that hits in the data cache.

The cache allocation policies are as follows:

- **Write allocation(WA):** On ARM, a cache line is allocated in case of a write miss. A store instruction triggers a burst read to occur. There is a line fill to obtain the data for the cache line, before the write is performed.
- **Read allocation(RA):** A cache line is allocated on a read miss.

The cache update policies are:

- **Write-back(WB):** A write updates the cache only and marks the cache line as dirty. External memory is updated only when the line is evicted or explicitly cleaned [4].

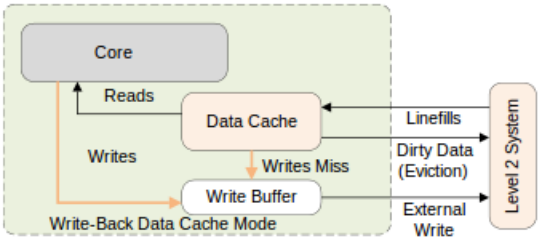


Figure 2.11: Cache Write-back

- **Write-through(WT):** A write updates both the cache and the external memory system [4]. This does not mark the cache line as dirty. Data reads which hit in the

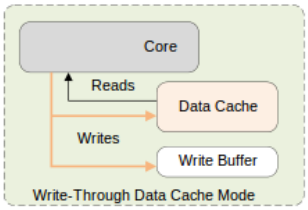


Figure 2.12: Cache Write-through

cache behave the same in both WT and WB cache modes.

Normal memory can be speculatively accessed by the processor and this means that it can potentially automatically load data into the cache without a programmer having explicitly requested a specific address. However, it is also possible for the programmer to give an indication to the core about which data is used in the future. The ARMv8-A provides preload hint instructions. It is implementation defined whether the caches support speculation and preload.

2.3.4 Configuring ARM Caches

ARM architectures [63] use the term flush and clean to depict the cache maintenance operations. They are described as follows:

Cache Flush: To "flush a cahche" is to clear it of any stored data. Flushing [63] simply clears the valid bit in the affected cache line. All or just portions of a cache may need flushing to support changes in memory configuration. The term invalidate is sometimes used in place of the term flush. However, if some portion of D-cache is configured to use a writeback policy, the data cache may also need cleaning.

Cache Clean: To "clean a cache" is to force a write of dirty cache lines from the cache out of main memory and clear the dirty bits in the cache line. Cleaning a cache [63] reestablishes coherence between cached memory and main memory, and only applied to D-caches using writeback policy.

2.3.4.1 Using Coprocessor 15 for cache maintenance

There are several coprocessor 15 registers [63] used to specifically configure and control ARM cached cores. Primary C15 registers: c7 and c9 control the setup and operation of the cache. Secondary CP15: c7 registers are write only. They clean and flush cache. The CP15: c9 register defines the victim pointer base address which determines the number of lines of code or data that are locked in the cache.

2.3.4.2 ARM Cache Maintenance Instructions

The ARM architecture offers following privileged operations to interact with caches and they can be executed using operations mentioned in Table: 2.2

- Invalidate (I & D-cache)
- Clean (D-cache)
- Clean + Invalidate (D-cache)
- Cache maintenance by Virtual Address
- Cache maintenance by Set/Way

Table 2.2: ARM Cache Maintenance Instructions

DCCMVAC	Clean data cache line by MVA to PoC
DCCSW	Clean data cache line by MVA to PoC
DCCMVAU	Clean data cache line by MVA to PoU
DCCIMVAC	Clean data cache line by MVA to PoU
DCCISW	Clean and invalidate data cache line by set/way

2.3.5 ARMv7 Cache Architecture

Our test device Nexus 5 has KRAIT 400 CPU which is based on Armv7 architecture. Some of the worth mentioning features of Armv7 cache architecture are as follows:

- (Modified) Harvard architecture
  - Multiple levels of caching (with snooping)
  - Separate Instruction (I) cache and Data (D) cache (no snooping between I cache and D cache)
  - Either PIPT or non-aliasing VIPT for D-cache
  - Meeting at the Point of Unification (PoU)
- Controlled by attributes in the page tables
  - Memory type (normal, device)

2.4 Operating Systems Caveats

2.4.1 Shared memory

Operating systems [77][11] use shared memory [55] (Figure: 2.13) to reduce memory utilization and for speed enhancement. For example, there could be several processes in the system running the bash command shell. It is not a good practice to have several copies of bash, one in each process 's virtual address space. It is better to have only one copy in physical memory and all of the processes running bash share it. Dynamic libraries are another common example of executing code shared between several processes. For instance, libraries used by several programs are shared among all processes using them. The operating system loads the libraries into physical memory only once and maps the same physical memory into the address space of each process. It enhances the execution speed as the code is kept only once in the memory, CPU caches and address translation units.

The operating system employs shared memory in several more cases. First, when forking a process, the memory is shared between the two processes. Only when the data is modified, the corresponding memory regions are copied. Second, a similar mechanism is used when starting another instance of an already running program. Third, it is also possible for user programs to request shared memory using system calls like *mmap()*. The operating system tries to unify these three categories. On Linux, mapping a program file or a shared library (Figure: 2.14) file as a read-only memory with *mmap()* results in sharing the memory with all these programs, respectively programs using the same shared library or program binary.

Sharing memory pages [77] [78][54] between non-trusting processes is a common method of reducing the memory footprint of mutli-tenated systems and can also be used in inter-process communication mechanisms in two co-operating processes. As pages can also

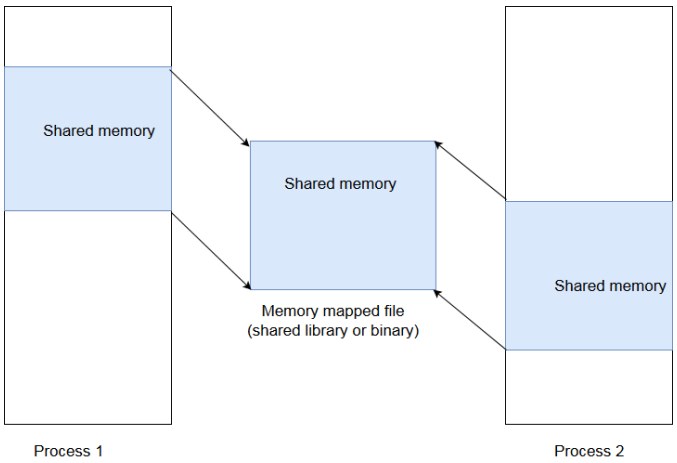


Figure 2.13: Shared Memory

be shared between non-coperating processes, the system must protect the content of the pages to prevent malicious processes from modifying the shared contents.

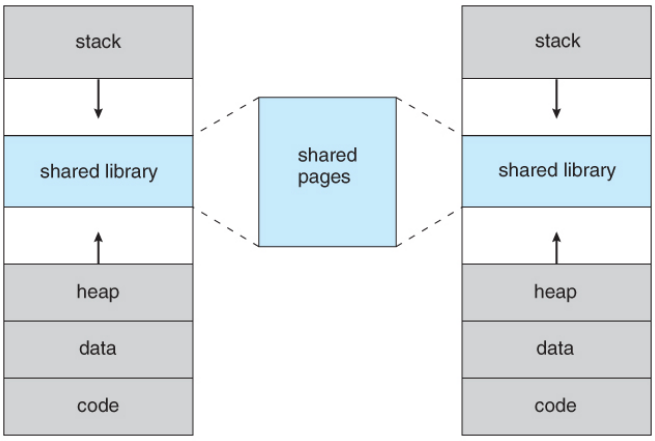


Figure 2.14: Shared library on an operating system

In order to protect the information, the operating system maps shared memory pages as copy-on write [77][78]. In case of a read operation on a shared page($copy-on-write$), a CPU trap is raised. The system software which gains control of the CPU during the trap copies the contents of the shared page, maps the copied page into the address space of the writing process and resumes the process. The idea behind $copy-on-write$ mechanism is to protect shared pages from modifications. However, the delay introduced while modifying a shared page can be detected by processes, leading to a potential information leak.

As Android is based on Linux, the concepts of shared memory remain the same.

However, Android applications are mostly written in JAVA and sharing memory in between spy process and victim process can be a bit difficult. Hence, we aim to target the shared libraries and binaries on Android. We will illustrate in the next chapter that shared memory [78] exposes processes to information leaks via cache access timings. Processes can retrieve information on virtual and physical address mappings using operating systems service like ($/proc/ < pid > /pagemap$) or ($/proc/ < pid > /maps$). Additionally, we can identify congruent addresses for eviction by using *pagemap* utility provided by Linux. An EVICT+RELOAD technique which is a cache-based side channel attack exploits all these weaknesses to monitor access to memory lines in the shared memory.

Understanding Side Channels on Android

3

Side channels [1] are quite a recent category of vulnerabilities in android smartphones. As described in Section: 1 information may unintentionally leak through radio signals, sensors, power consumption or through the state of a processor's memory cache. Thereby, it empowers a malicious application to spy on a victim application by observing these channels of information on a smartphone.
Let us take an example of Battery Manager class in android. Battery Manager is mostly used to keep a check on the battery consumption of an android smartphone. Interestingly, it can be exploited as a potential side channel. For instance, battery level information can be used to identify the following:

- Loading time of an application
- Run time of the application on the processor etc.

In addition to battery levels, android devices can also report their instantaneous current and voltage readings to the users through Battery Manager Class. However, the frequency at which these readings are communicated to the user is limited by the hardware. Also, the scenario varies from device to device as fuel gauges present in certain devices (e.g. Samsung Galaxy S4) do not provide instantaneous current readings owing to security measures. But on the other hand, fuel gauge *MAX17048* present on Nexus 5 reports instantaneous current and voltage readings (written by I2C) through *current_now* and *voltage_now* files respectively:

Battery parameters	**File Location**
Instantaneous Current	/sys/class/power_supply/battery/current_now
Instantaneous Voltage	/sys/class/power_supply/battery/voltage_now

Table 3.1: Instantneous Current and Voltage Files in Android

These two files can be accessed on our test device Nexus 5 without any root privileges. In our following experiment, we aim to exploit this potential source of leakage on Nexus 5 via a spy application (rogue app) running in parallel to a crypto application (victim app). Our two applications will carry out the demonstration of Power Side Channel(PSC) attack on Nexus 5. The victim application performs RSA on a user input for around 10000 iterations. Victim app is triggered by our rogue app which accepts the input from the user and raises an intent for victim app to perform RSA operation on the user input. During this time, the rogue app monitors the battery status and logs the instantaneous current and voltage values by polling *current_now* and *voltage_now* files. The idea is to collect traces where we can identify square and multiply operation in RSA implementation and

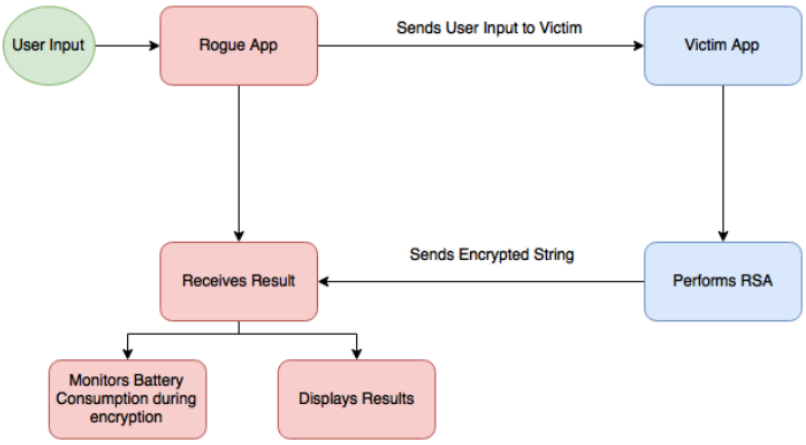


Figure 3.1: Experimental Set-up

then recover the secret key. One pre-selected secret key is used to perform RSA on different user inputs.
Before and after consumption of battery levels does not give sufficient meaningful information as it lacks precision. Hence, it can be safely concluded that without any hardware assistance it is not a significant exploit. However, instantaneous voltage is reported in milli volts(mV) and instantaneous current is reported in milli amperes(mA). Through the graphical representation of (as shown in Figure: 3.2) the instantaneous current readings of our rogue app w.r.t time, we can identify the time duration for which our app was scheduled on cpu, in addition to the peaks which can reveal the keystrokes. Since the processor operates at 2.24GHZ and the fuel gauge operates at 400khz, there is a limit (as shown in Figure: 3.3) to the number of readings which we can get from the device (approximately 25 readings per $10^{-}7$ sec).
Hardware-based identification and exploitation of this side channel can still be taken into consideration as they will not be limited by the speed of the I2C bus.

3.1 Microarchitectural Attacks

In addition to PSC attacks, there exist microarchitectural attacks [16] which exploit microarchitectural functionalities of processor implementations. They can compromise the security of computational environments even in the presence of sophisticated protection mechanisms like virtualization and sandboxing. There lies an inherent gap in between the current processor architectures and the ideal secure computing environment. This is further accentuated by the loopholes in the operating system managing various processor resources. These attacks exploit the microarchitectural components and functionalities of a processor to reveal cryptographic keys. The functionality of some processor components generates data dependent variations in execution time and power consumption during the

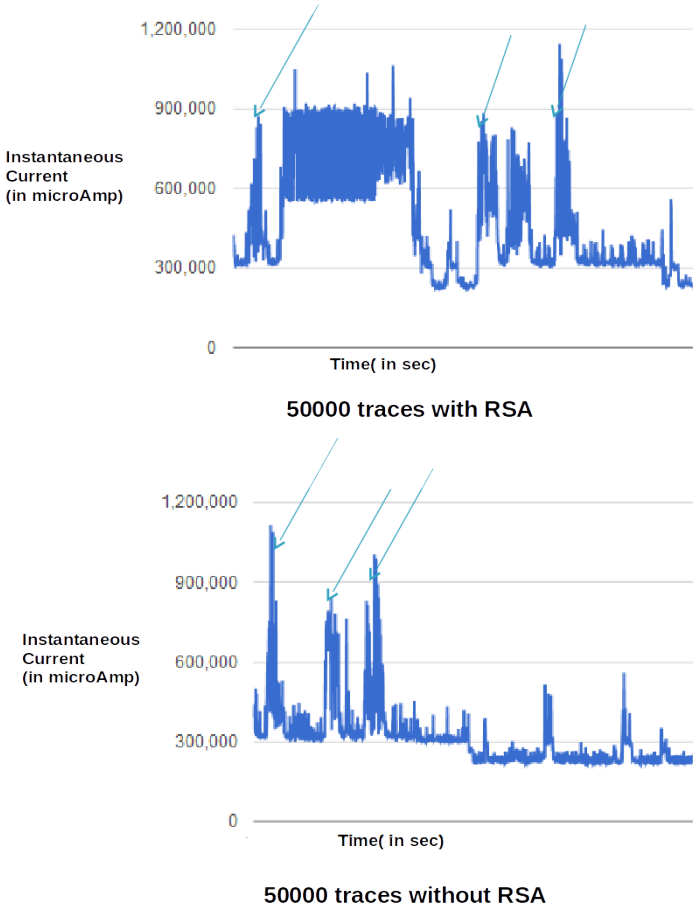


Figure 3.2: Arrows Depicting Keystrokes after and before RSA operation

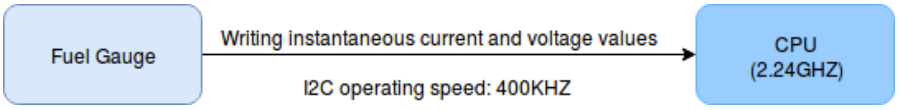


Figure 3.3: Fuel Gauge and processor interaction

execution of cryptosystems. These variations either directly give the key value out during a single cipher execution or leak information which can be gathered during many executions and analyzed to compromise the system. Microarchitectural covert channels can be based on **L1 cache, Branch Predictor Unit, Last Level Cache and Memory**

build

Bus. According to the literature, there are two types of Microarchitectural attacks [16]:

- Cache Analysis
- Branch Predictor Analysis

3.1.1 Cache Attacks

We have chronologically discussed the cache attacks in the Section: 1.1. In this section we aim to discuss cache attacks with respect to their applicability and exploitation on different architectures (especially ARM). A cache attack exploits the cache behavior of a cryptosystem by obtaining the execution time and/or power consumption variations generated via cache hits and misses. Cache analysis techniques enable an unprivileged process to attack another process, e.g., a cipher process, running in parallel on the same processor as done in past research work(Section: 1.1). Furthermore, some of the cache attacks can even be carried out remotely, e.g., over a local network. The previous cache attacks are data-path attacks, i.e., exploit the data access patterns of a cipher. The memory accesses of software cryptosystems, especially S-box based ciphers like DES and AES, employ key-dependent table lookups, indices of which are simple functions of the key and the plaintext. Revealing these memory access patterns, i.e. lookup indices via cache statistics and the knowledge of the processed message, e.g. in a known-text attack, makes it relatively easy to break these ciphers.

In this work of research, we discuss the access driven cache timing attacks. As the name of cache timing attacks suggests, they utilize the particularities of microcontrollers and microprocessors with the cache memory which frequently exhibit key-dependent timing. Cache timing attacks on many block ciphers with S-boxes become possible since S-box invocations in software are often implemented as indexed table look-up operations that can require different execution times for different inputs due to RAM cache hits and misses. When the inputs to S-boxes are key-dependent, this timing information frequently turns out sufficient to recover the entire key. Timings attacks are further classified into time driven, trace driven and access driven attacks which are discussed as follows:

- **Trace-driven attacks:** For these attacks a detailed cache profile based on the information of every single memory access is necessary, i.e. for every look-up operation an attacker knows whether it resulted in a cache hit or a cache miss. The performance counters of modern CPUs might be used to establish such a memory-access profile. Bertoni et al. [21] work is a nice contribution to this category of attacks.
- **Access-driven attacks:** The purpose of these attacks is to determine which cache lines or cache sets have been accessed during the encryption. Hence, knowledge of the location of the precomputed S-boxes or T-tables within the memory as well as information about the cache architecture is necessary. However, fewer measurement samples are necessary than in the case of time-driven attacks. Historically [58][59], these attacks can be further divided into following categories:

  - **EVICT + TIME:** The attacker measures the time it takes to execute a piece of victim code. Then attacker flushes part of the cache, executes and times the victim code again. The difference in timing tells whether the victim uses that part of the cache.
  - **PRIME + PROBE:** The attacker accesses memory to fill part of the cache with his own memory and waits for the victim code to execute. This is called the Prime Step. Then the attacker measures the time it takes to access the memory that he carefully placed in the cache before. This is called the Probe Step. If the access time is higher than a certain threshold for certain cache line, then we know that the victim process evicted those cache lines from the cache. If the access time is less than a certain threshold, then it becomes clear that victim did not access those lines or evict those cache lines.
  - **FLUSH + RELOAD:** The flush and reload attack utilizes the fact that processes often share memory. By flushing a shared address, then wait for the victim and finally measuring the time it takes to access the address an attacker can tell if the victim placed the address in question in the cache by accessing it.

[1]

- **Time-driven attacks:** These attacks require only minor knowledge of the implementation and the hardware architecture under attack [23]. Depending on the provided input the implementation might leak different timings [57]. Thus, the basic idea of time-driven attacks is to gather timing information of many encryptions and to perform statistical correlations in order to recover the used secret key. Attacks in this category typically require far more measurement samples than attacks within the previously mentioned two categories. One of the landmarks in this category of attack is Bernstein's cache timing attack on AES [20].

  The rising popularity of smartphones in our everyday life clearly states the need for the investigation of such cache attacks on modern smartphones in a realistic scenario. It also becomes important to study the assisting techniques which make these attacks more viable in realistic scenarios. In the next chapter we will discuss the implementation steps of an access-driven cache timing attack on OpenSSL AES implementation.

3.2 Exploitable ARM properties for Cache Attack

ARM devices implement a technique called Amba Cache Coherent Interconnect(ACCI) that facilitates fast interprocessor connections very similar to HyperTransport protocol in AMD processors or Intel QuickPath Interconnect Technology. This technology helps to maintain cache coherency across ARM CPUs using snoop filter protocol supported by cache directory architecture. Thus upon a shared memory read miss, the snoop

[1] There are some other variants as well like FLUSH+FLUSH and EVICT+RELOAD. They are discussed in the next chapter

filter checks whether the same memory block is cached in an adjacent processor. If succesful, a direct cache-to-cache link will be established thereby eliminating the need for a slow DRAM access. We exploit this functionality by using *libflush* to attack AES implementations on ARM. We will implement the EVICT+RELOAD attack on AES implementation of OpenSSL 0.9.7a and OpenSSL 1.0.1g to extract the full key.

3.3 AES

The Advanced Encryption Standard (AES) [30] was introduced in 2001 by the National Institute of Standards and Technology (NIST). In the year 1997, it was announced that Data Encryption Standard (DES) has become vulnerable to brute force attacks and there is a need for much more advanced symmetric key algorithm. AES is a result of a competition which started in 1997 and ended in 2000, where it was selected from various candidates. AES is a subset of **Rijndael Cipher** proposed by J. Daemen adn V. Tijmen. AES is today's most commonly used block cipher in the Internet and software Market: applications include disk and file encryptions, wireless LAN security, IPSec (standard for securing Internet protocol at the network layer), Transport Layer Security (successor of SSl), VoIP security, smart cards, microprocessors and many others.

3.3.1 Description

AES is a block cipher with 128 bit (16 bytes) input represented as follows:

$$p = (p_0.....p_{15})$$

The key size can be of 128, 192 or 256 bits.

$$k = (k_0.....k_{n-1})$$

Here n can be 16, 24 or 32 depending on the size of the key. For our discussion we will use n=16, however the explanation can be extended to longer keys as well. AES is a key-iterated block cipher: where it is composed of a key schedule and repeated round transformations as described in Figure:3.4. An encryption of plaintext **p**is with key **k** results into a ciphertext **c** which can be denoted as follows:

$$c = E_{AES}(p, k)$$

In the key scheduling process, a 16-byte key k is expanded into 11 round keys $K^{(r)}$ where each round is denoted by $r=\langle 0, ..., 10\rangle$. Round keys are calculated as follows :

$$k = (k_0, ..., k_{15}) \rightarrow K^{(r)} = (K_0^{(r)}, ..., K_{15}^{(r)})$$

In the first round the complete key k is used as round key which is K^0. An important point here is that **Key expansion** is irreversible and if one of the round keys is known then any other round key can be computed. After first `AddRoundKey`operation, AES performs 10 rounds of `SubBytes`,`ShiftRows`, `MixColumns` and `AddRoundKey` on a state. There is no MixColoumn operation in the last round.

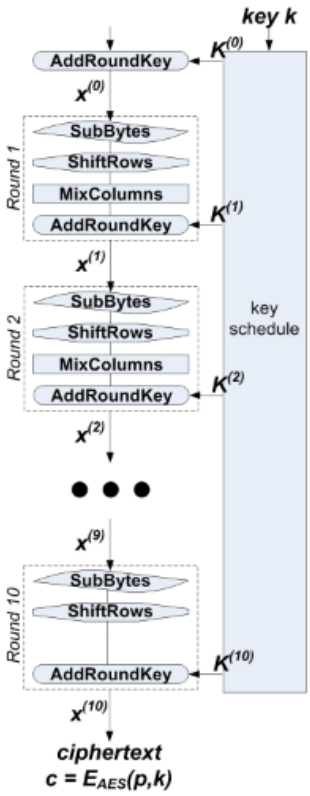


Figure 3.4: AES Encryption

3.3.2 OpenSSL AES Implementation

OpenSSL AES implementation makes use of pre-computed T-tables which is susceptible to cryptanalysis. These large look-up tables are susceptible to cache timing attacks, cache probing attacks and cache collision attacks [50][58]. The cipher can be directly computed using simple lookups and bitwise XOR operations. Several such tables are precomputed once by the programmer or during system initialization. In OpenSSL 0.9.7a AES, there are 10 such tables T_0, T_1,T_2, T_3, T_4(only used in last round), $T_d^{(0)}$, $T_d^{(1)}$, $T_d^{(2)}$, $T_d^{(3)}$,$T_d^{(4)}$ each containing 256 4-byte words. On the other hand OpenSSL 1.0.1g AES, there exists 9 such tables T_0, T_1,T_2, T_3,$T_d^{(0)}$, $T_d^{(1)}$, $T_d^{(2)}$, $T_d^{(3)}$,$T_d^{(4)}$. In this AES implementation T_0,T_1,T_2,T_3 are used in the last round. The contents of the tables are not significant for purpose of attack.

During key-set-up, a given 16-byte secret key $k = (k_0.....k_{15})$ is expanded into 10 round keys $K^r = (K_0^{(r)}, K_1^{(r)}, K_2^{(r)}, K_3^{(r)})$. The 0^{th} round key is just the original key $K_j^{(0)} = (k_{4j}, k_{4j+1}, k_{4j+2}, k_{4j+3})$for $j = 0, 1, 2, 3$. The details of the rest of the key expansion can be omitted for this document.

Given 16-byte plaintext p = $(p_0, p_1, ..., p_{15})$, encryption proceeds by computing a 16-byte intermediate state $x^r = x_0^{(r)}, ..., x_{15}^{(r)}$ at each round r. The initial state x^0 is computed by $x_i^0 = p_i \oplus k_i$ and then, the first 9 rounds are computed by updating intermediate state as follows, for r = 0,.....,8:

$$
\begin{aligned}
x_0^{(r)} &= T_0[x_0^{(r-1)}] \oplus T_1[x_5^{(r-1)}] \oplus T_2[x_{10}^{(r-1)}] \oplus T_3[x_{15}^{(r-1)}] \oplus K_0^{(r)} \\
x_1^{(r)} &= T_0[x_4^{(r-1)}] \oplus T_1[x_9^{(r-1)}] \oplus T_2[x_{14}^{(r-1)}] \oplus T_3[x_3^{(r-1)}] \oplus K_1^{(r)} \\
x_2^{(r)} &= T_0[x_5^{(r-1)}] \oplus T_1[x_{13}^{(r-1)}] \oplus T_2[x_2^{(r-1)}] \oplus T_3[x_7^{(r-1)}] \oplus K_2^{(r)} \\
x_3^{(r)} &= T_0[x_{12}^{(r-1)}] \oplus T_1[x_9^{(r-1)}] \oplus T_2[x_6^{(r-1)}] \oplus T_3[x_{11}^{(r-1)}] \oplus K_3^{(r)}
\end{aligned}
$$

Thus, for each one byte input, T tables will provide with a 4-byte output. The S-box Tables T_i are generated from two constant 256 byte-tables, designated as S and S' as can be seen in Figure:(3.5) and (3.6)

lsb / msb	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
10	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
11	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
12	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
13	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
14	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
15	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3.5: S-box S

lsb / msb	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	c6	f8	ee	f6	ff	d6	de	91	60	02	ce	56	e7	b5	4d	ec
1	8f	1f	89	fa	ef	b2	8e	fb	41	b3	5f	45	23	53	e4	9b
2	75	e1	3d	4c	6c	7e	f5	83	68	51	d1	f9	e2	ab	62	2a
3	08	95	46	9d	30	37	0a	2f	0e	24	1b	df	cd	4e	7f	ea
4	12	1d	58	34	36	dc	b4	5b	a4	76	b7	7d	52	dd	5e	13
5	a6	b9	00	c1	40	e3	79	b6	d4	8d	67	72	94	98	b0	85
6	bb	c5	4f	ed	86	9a	66	11	8a	e9	04	fe	a0	78	25	4b
7	a2	5d	80	05	3f	21	70	f1	63	77	af	42	20	e5	fd	bf
8	81	18	26	c3	be	35	88	2e	93	55	fc	7a	c8	ba	32	e6
9	c0	19	9e	a3	44	54	3b	0b	8c	c7	6b	28	a7	bc	16	ad
10	db	64	74	14	92	0c	48	b8	9f	bd	43	c4	39	31	d3	f2
11	d5	8b	6e	da	01	b1	9c	49	d8	ac	f3	cf	ca	f4	47	10
12	6f	f0	4a	5c	38	57	73	97	cb	a1	e8	3e	96	61	0d	0f
13	e0	7c	71	cc	90	06	f7	1c	c2	6a	ae	69	17	99	3a	27
14	d9	eb	2b	22	d2	a9	07	33	2d	3c	15	c9	87	aa	50	a5
15	03	59	09	1a	65	d7	84	d0	82	29	5a	1e	7b	a8	6d	2c

Figure 3.6: S-box S'

The S-box tables are computed as follows :

$$
\begin{aligned}
T_0 &= (S', S, S, S \oplus S'), \\
T_1 &= (S \oplus S', S', S, S),
\end{aligned}
$$

build

$$T_2 = (S, S \oplus S', S', S),$$
$$T_3 = (S, S, S \oplus S', S')$$

These pre-computed tables provide for a significant increase in performance as each round is replaced by table lookups and bitwise xor operations. However, memory consumption increases as these tables consume equal to or more than 8KB of memory.

3.4 Correlation Analysis

Correlation Analysis [65] is a statistical technique which is very popular in the case of Differential Power Analysis (DPA). The reason behind its popularity can be attributed to the ability of this attack to extract the secret key even from the noisy measurements. The idea is to reveal the secret key using a large number of traces that have been recorded while the devices encrypt or decrypt different data blocks. It exploits the data dependency of the traces (for e.g power consumption traces) on the devices. The general strategy of correlation analysis can be enumerated as follows:

1. **Choose an appropriate Intermediate Value for an encryption algorithm:** The intermediate value (y) should be a function of a data set ($D = d_1..d_D$) containing random values which can be either plaintext or ciphertext and a part of key value(k). Such intermediate values can be used to reveal k.

$$y = f(d, k)$$

2. **Measurements during encryption/decryption of data blocks:** The next step to measure the power consumption or some other form of leakage with respect to the encryption/decryption of data set (D). In the case of power consumption, we can measure the power for a certain amount of time T, and a power trace represents the power consumptions of operations on data block D for trace length T which is ($t_{i'} = (t_{i,1}..t_{i,T})$. It is worth mentioning that in the case of power traces, alignment of traces is a substantial issue. Usually a better triggering mechanism can resolve such an issue or techniques like Dynamic Time wrapping etc can also be used.

3. **Calculate Hypothetical Intermediate Values:** This step requires to calculate a hypothetical intermediate value for every possible choice of k. For e.g if k is one byte in size, there will be 256 possibilities or key hypothesis to it. A vector k can be descibed as follows:

$$k = \langle k_1..k_{256} \rangle.$$

Hypothetical intermediate values are calculated as described in the first step for all D data values and all k Key Hypothesis. A new vector V is generated which is as follows:

$$v_{i,j} = f(d_i, k_j) \text{ for i=1,..,D and j=1,..,K}$$

build

The goal of our analysis is to find which column of V is processed during D encryption/decryption runs using k_j key hypothesis.

4. **Map Intermediate Values to Actual Measurements:** The next step is to use simulation techniques to map the hypothetical intermediate values to our measured values. For each hypothetical intermediate value a hypothetical measurement value is generated. Several leakage models can be used for this purpose. In case of DPA , Hamming Weight and Hamming Distance are the most common leakage models. There are other leakage models in existence as well such as Bit Model, ID Model and Zero-Value Model. We are performing cache attacks where we can no longer use Hamming Weight or Hamming Distance model. We will make use of **Bit Model (testing against every single bit in the key), ID model/Zero-Value model**. The quality of simulation is also based on the amount of knowledge an attacker has about the device under attack.

5. **Compare the Actual traces with the Hypothetical Traces:** In this step the adversary compares the hypothetical measured values of each key hypothesis with the recorded traces at every position.

All these steps are core to our attack implementation as they exploit even the smallest dependancy in between hypothetical traces and the recorded traces.

3.5 Key Rank Estimation

Side-channel attacks are a vital aspect of a security evaluation framework. Interpreting the results to give a security level is however not always so easy. Rank estimation algorithms give a method to compute an interval for the rank of a key and this can be used to estimate the security level of an implementation.

Suppose we have an implementation that uses a key $k*$. Given side-channel attack results on this implementation, the rank of $k*$ is defined as the number of keys k that have a higher probability of being the one used in the implementation according to the side-channel results. A higher rank therefore means less information leakage and a more secure key.

Key enumeration [70] and rank estimation [61] algorithms have recently emerged as an important part of the security evaluation of cryptographic implementations, which allows post-processing the side-channel attack outcomes and determine the computational security of an implementation after some leakage has been observed. In this respect, key enumeration can be seen as an adversarial tool, since it allows testing key candidates without knowledge of the master key. By contrast, rank estimation as an evaluation tool since it requires the knowledge of the master key. Its main advantage is that it allows efficiently gauging the security level of implementations for which enumeration is beyond reach (and therefore are not trivially insecure).

Concretely, state-of-the-art solutions for key rank estimation are essentially sufficient to analyze any (symmetric) cryptographic primitive. These Algorithms typically allow estimating the rank of a 128- or 256-bit key with an accuracy of less than one bit, within seconds of computation. By contrast, efficiency remained a concern for key enumeration algorithms for some time.

3.6 An overview of White-Box

In black box cryptosystems, the specification is not generally available to the user and is generally not appreciated. It is because when these cryptosystems were reverse engineered and analysed, it was shown that they were weak and easy to break. Hardware Security Modules (HSMs) such as Trusted Platform Modules (TPMs) or smart cards are some of the common approaches to protect the secret key. However HSMs can be considered as grey boxes as they may leak information in the form of side-channels. They are also far less flexible than software implementations. Cryptography implementations in softwares which guarantee the security of secret key while being run in an environment controlled completely by an attacker are called white box implementations. Several academic papers [75][24][31] have described inner workings of white-boxes. In 2015 Bos et.al [24] showed that publicly available white-box implementations are highly vulnerable to an attack called *Differential Computation Analysis*. This attack was against memory access traces captured from executions of a white-box. On Black Hat Europe 2015, Saneflix et al. showed another attack based on *Differential Fault Analysis*. Our work is influenced by Differential Computation Analysis (DCA) of cache traces instead of memory traces and deploying it on android is one of the goals of this research. DCA requires less number of traces in comparison to Differential Power Analysis (DPA) techniques and requires less time in running the attack. The reason could be attributed to the fact that DCA traces do not contain signal noise, so values can always be observed precisely. Chapter 4 gives further information about side-channel attacks against a custom white-box implementation on an android smartphone.

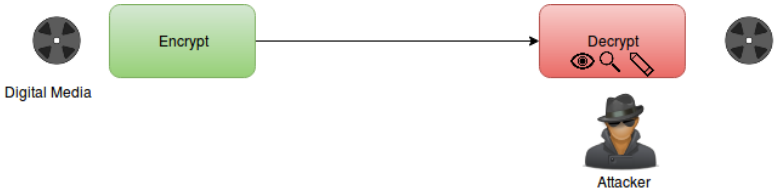


Figure 3.7: White-box attack

3.6.1 White-Box on an Android Platform

White box cryptography [12] is becoming very popular for both mobile payments and digital media as it achieves the necessary security goals like keeping money and payment information of the involved parties secure and protecting intellectual property rights at

the same time. It is cheaper and viable alternative to HSMs. Since HSMs are very expensive and harder to deploy than white boxes, their technical and user acceptance is also relatively low. Figure: 3.7 describes a White-Box threat model which has following attributes :

- The attacker can observe the encryption process from within the system
- The attacker can modify anything at will, including the cryptographic algorithm

4 Case Studies

4.1 Introduction

In the past few decades, we have seen a huge leap in the circuit fabrication techniques. Such an advancement has led to the addition of caches to fill the gap processor-memory performance gap as discussed in Section:2 and have mitigated the effects of memory latency to a huge extent. However, it comes with its own set of disadvantages as the improvement in average performance due to caches comes at the expense of a vastly increased variability in performance. This has been known for many years to cause problems in the design of safety-critical real time systems where it is imperative that a series of deadlines be satisfied even as the presence of caches makes it very difficult to determine the worst-case performance time [60]. In the recent years, it has been shown that the presence of caches and the resulting timing variability makes possible a number of cryptanalytic side channel attacks [25][46][44][39] [71][38][64]. In this chapter, we will describe various cache attack scenarios ranging from an attack on an artificial application to the attack on a crypto application. Such attacks enable a third-party process/application to exploit CPU data cache, which in turn, can be used to infer details about the data that was being processed or application being run.

Most of the research conducted on cache attacks in the past dealt heavily with Intel x86 architectures. Nowadays, ARM is one of the most popular choices because smartphones, tablets, and many IoT devices are built on ARM. Given this widespread application of ARM processors, researchers [53] [64][38][37] have started exploring the feasibility of cache attacks on ARM.

4.2 Cache Attacks

Broadly cache attacks are either based on the cache storage attacks or cache timing attacks. Cache timing attacks make use of a simple model to correlate the execution time of an algorithm with the state of the cache used by the CPU in charge. It is assumed that the execution time is lower if the data needed by the algorithm is already stored in a cache line (cache-hit). On the other hand, if the required data is not present in the cache and hence has to be loaded from the main memory (cache-miss), this will result in a longer execution time. This model is simple, but reasonable and only relies on the cache architecture of the CPU. Weiss et al. [71] provide a suitable attack scenario where they consider Multi-Core Aspects in Virtualized Embedded Systems. Time-driven attacks require a large number of plain-texts and cipher-texts to perform statistical analysis while might be a hurdle in a realistic scenario. Hence, our focus is on the access-driven attack as they provide with promising results even in noisy enviornments.

Also, the number of encryptions required to perform access-driven cache attack is way lower than time-driven cache attack.

As a part of our case studies, we perform access driven cache attack on quad-core Krait processors (Krait 300 and Krait 400). We infer the information about the memory accesses being performed during an encryption. We introduce a novel strategy which is a combination of strategies proposed by Osvik et al. [58],and Spreitzer et al. [64] to know about the cache sets accessed before/during/after encryption. Eventually, we aim to recover the entire secret key used for encryption. Further, we extend the application of our attack to white-box cryptosystems. We demonstrate a cache attack on a custom white-box implementation of AES by Chow et al.[29]

4.2.1 Adversary Model

As discussed in 4.2 our focus is on access-driven cache attack. These attacks operate on cache memory that is shared between processor cores on an Android smartphone. We assume that the adversary is a legitimate user of the android smartphone and is able to install and launch the applications or facilitate native code execution. The attacker is assumed to be able to observe the cipher-text and plain-text (not necessarily required) and has the ability to interact with the system performing the crypto operation. We will also exploit hardware performance counters to achieve an accurate timing measurement. The adversary should have root access only once to install the kernel module. We are considering PRIME+ PROBE and EVICT+RELOAD cache attacks on a full blown operating system-Android 5.0.1 running on our test devices (1.6). It is also assumed that while attacking a crypto application, an attacker is able to observe the time required to access a cache set before and after eviction used in encryption.

4.2.1.1 Additional Criteria to take into consideration

In addition to the high-level view of the attacker 's model, we would like to mention some criteria which are equally important for successful deployment of our attacks.

1. **Curious Case of Processor optimizations:** Our motive is to be able to identify the cache sets being accessed by the victim process during execution. However, several processor optimizations may result in false positives due to speculative memory accesses issued by the victim's processor. These optimisations include data prefetching to exploit spatial locality and speculative execution. Therefore, while analyzing the attack results the attacker must be aware of these optimisations and develop strategies to filter them (discussed in Chapter-A). Also. the prefetching mechanism is poorly documented which further adds to the attacker 's grievances. In our case, most of the times the tables are present in the cache memory and hence the probability of prefetcher getting triggered gets reasonably reduced.

2. **Synchronization of victim and spy process:** As victim access is independent of the execution of the spy process code, increasing the wait period reduces the probability of missing the access due to an overlap. On the other hand increasing the

waiting period reduces the granularity of the attack. Our attack is a cache timing attack where we are monitoring the effects on the cache state corresponding to our attacker process and victim process, both. Hence we should see that the attacker process does not get scheduled for most of the processor time and the victim process doesn't get scheduled at all or most of the time is out of the running queue. To avoid such a scenario proper synchronization between victim and attacking process should be established. **In our scenario we have made use of file locks and *sleep()* system call**. It has made our implementation a bit slower (in the kernel source of Android 5.0, Semaphores and Mutexes are not implemented).

3. **Resolution of the attack:** One of the ways to improve the resolution of attack without increasing the error rate is to target memory accesses that occur frequently such as a loop body. But in such a case the attack will not be able to differentiate between the separate accesses.

4. **Address Layout Randomization:** When a page is shared, all the page entries in the virtual address spaces of the sharing processes map to the same physical page. As the LLC is physically tagged, entries in the cache depend only on the physical address of the shared page with no dependency on the virtual addresses in which the page is mapped. Consequently, we do not need to take care of the virtual to physical address mapping and the attack is oblivious to Address Space Layout Randomization (ASLR).

5. **Statistical Analysis:** An access-driven cache-timing trace, as described in the later sections is interpreted as a sequence of cache hits and misses on a per cache set or line basis. We have to take into consideration that these hits and misses are based on the memory access timings being above or below a certain threshold. Such calculations are quite sensitive to a particular processor, cache organization, type of operating system and load on the system. Hence, these readings are not normally error free. It is therefore necessary to perform some statistical analysis on our trace sets and take as many readings as possible.

4.2.2 Modeling the Cache Timing Behavior

The first step towards identifying cache hit/miss ratio is to model the cache timing behavior. It is a necessary step in order to find a correlation between the memory addresses being accessed and the ideal data. As cache eviction is another key aspect of a cache attack, we can identify a suitable eviction strategy provided an accurate timer is at our disposal. Therefore, in order to identify which is the most suitable eviction strategy for the device(1.6) under consideration, we require access to a very accurate timing mechanism which precisely measures the cache hits and misses.

Firstly, we allocate a large data structure (equal to or more than the size of L1 cache) of our target device. The next step is to access a particular address or set of addresses from the allocated ones in the cache and calculate the timing. Later, we will evict that address or addresses from cache and then will try to access them again (either

it will be a remote core fetch or a memory access). The timing difference between a cache-hit or a miss can be calculated using various available timing interfaces or instructions.

Unfortunately, **ARMv7** instruction set does not provide with an instruction like *rtdsc* unlike **x86** which can be used to calculate the CPU cycles for a cache miss or a cache hit. The user mode access to performance counters like (PMUSERNR) and cycle count register (PMCCNTR) is not possible by default. In order to calculate the number of CPU cycles passed during a memory access, we need to install a kernel module. The accuracy of these timing interfaces may have a huge impact on our results. In our case, we have installed a kernel module which allows access to PMCCNTR register. It is worth mentioning that we are using a custom kernel. The reason behind a custom kernel is the absence of loadable module functionality in the stock kernel. We also have root access so that we can load or unload our kernel module. One can access the timing interface using *perf* or *monotonic clock* which may or may not provide cycle accurate timings. Eventually, with a suitable threshold, we can identify the cache hits/ cache misses clearly.

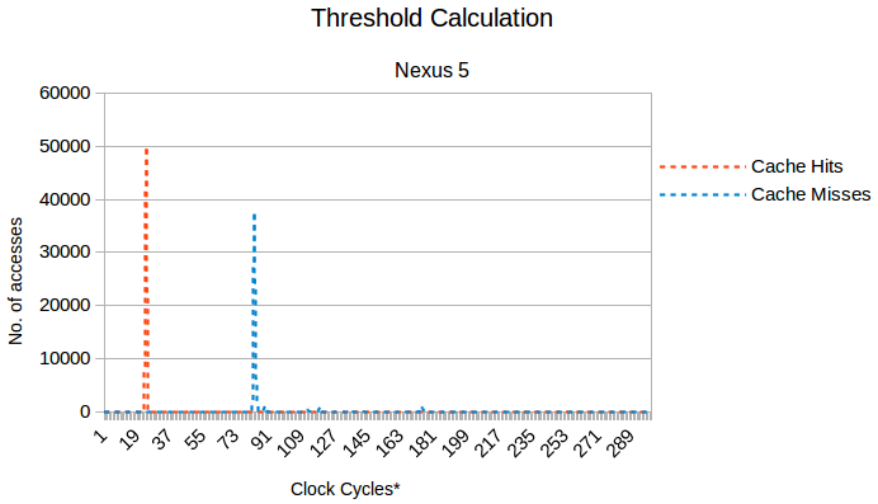
Figure: 4.1a and Figure: 4.1b demonstrate the suitable thresholds to identify a cache hit or a cache miss. X-axis defines the number of accesses to an address and Y axis defines the number of CPU cycles required to access an address. We have two peaks, where one signifies the maximum number of hits and the other signifies the number of cache misses once the corresponding cache line is evicted. The cache hits and cache misses are calculated on the basis of CPU cycles consumed by them. A suitable threshold can be calculated using this information which helps in a clear cut differentiation of hits and misses. Once a suitable threshold is calculated, we need to come up with a suitable eviction strategy to get a 100% eviction rate for a cache set/cache line in unprivileged mode.

Depending on the source of timing, the results may vary and sometimes they can be incorrect too. In the first place, we started our timing measurement of cache eviction using *perf* interface but the results were incorrect as all the cache hits and misses were at the same address or for the same bin value. We switched from *perf* to the *monotonic_clock* to see whether our eviction works or not and to our expectations it worked. Still, monotonic clock is not the best source to get accurate CPU cycles as it too gets affected by time slewing.
ARMv7 instruction set does not provide with unprivileged cache flush instructions (2). Hence, in order to perform fast and efficient eviction following points are to be taken into consideration:

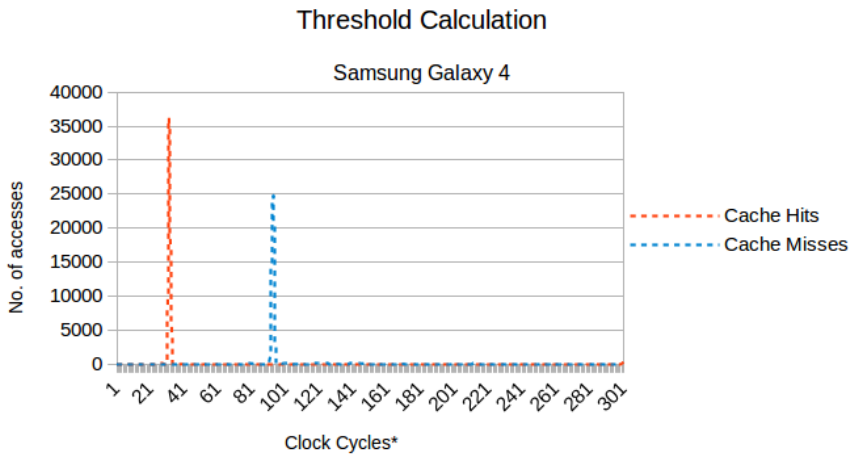
1. We need to find physically congruent addresses (addresses mapping to the same cache set) so that we can evict the desired cache lines.

2. We should be able to counteract pseudo-random replacement policies for L1 and L2 by running various access patterns and evaluate corresponding eviction strategies.

[0]Clock Cycles* = X-axis value *5



(a) Threshold calculation on NEXUS 5 using cycle counter



(b) Threshold calculation on Samsung Galaxy 4 using cycle counter

3. We should be able to distinguish between cross-core fetches and memory accesses based on CPU cycles

Fortunately, *libflush* [53] provides with a *Cache Eviction Strategy Calculator* which can resolve all the above mentioned issues. *Cache Eviction Strategy Calculator* tool helps us to calculate how many addresses are to be evicted (N), what should be the loop size (A), what should be the step size for the loop (D) etc, to reach a perfect eviction rate. The calculated eviction strategy is used while building the spy application. The cache eviction strategy is based on the eviction algorithm in [39] :

build

4.3 Notations

Let the size of cache line on our test devices [1.6]be represented by δ which is 64 bytes. Let the number of elements in each cache line be, σ, which is as follows:

$$\sigma = \frac{Total\ number\ of\ elements\ *\ size\ of\ elements}{\delta} \tag{4.1}$$

for e.g there are 256 4-Bytes elements in a table than the number of elements in each cach-line would be 16

4.3.1 EVICT + RELOAD Attack

EVICT+RELOAD is a variant of FLUSH+RELOAD as instead of flushing a cache line, we perform eviction. Once a suitable threshold and appropriate eviction strategy is calculated using an accurate timing mechanism, the next step is to decide the other attack parameters. We demonstrate the implementation of Evict+Reload attack on Nexus 5 and Samsung Galaxy S4. in the next chapter. It is one of the most powerful cache attacks and exploits one of the operating systems functionalities like usage of shared libraries. Below mentioned is the algorithm for EVICT+ RELOAD technique. Figure: 4.2 describes the attack strategy.

Algorithm 1: Algorithm for EVICT + RELOAD cache attack

1 Map a shared library or binary as a shared object into attacker's address space
2 Evict a cache line (instruction or data) from the shared memory
3 Schedule victim process to check if it loads or not the evicted cache line
4 The attacker process checks if evicted cache lines are loaded by victim program

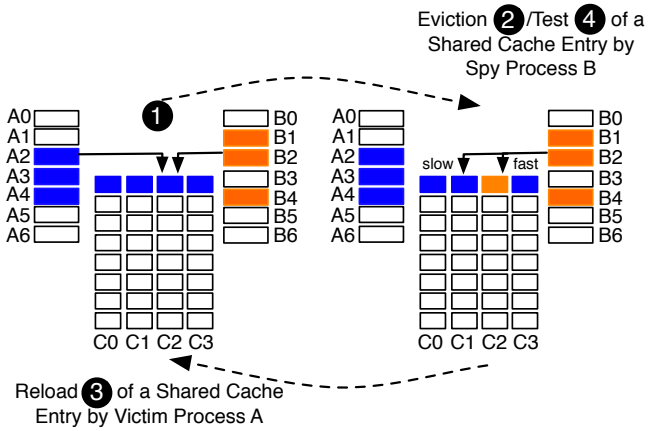


Figure 4.2: EVICT+RELOAD cache attack

build

4.3.2 PRIME + PROBE Attack

We also demonstrate the implementation of the PRIME+PROBE attack on Nexus 5 and Samsung Galaxy S4. It does not involve the usage of shared libraries. Below mentioned is the algorithm for PRIME + PROBE technique. Figure: 4.3 describes the attack strategy.

Algorithm 2: Algorithm for PRIME + PROBE cache attack

1 Attacker primes the cache lines
2 Victim process evicts the cache lines during encryption
3 The attacker process checks data to determine if the primed sets were accessed or not

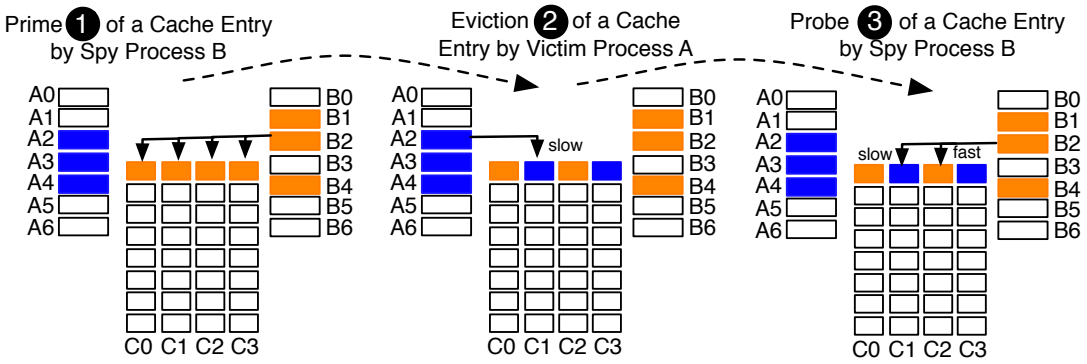


Figure 4.3: PRIME+PROBE cache attack

4.4 Attack Scenarios

4.4.1 Cache Attack on a Shared Library

In the wake of understanding the scope of cache attacks, we begin with an elementary attack on an application using shared library. We aim to spy on the application by mounting EVICT+RELOAD attack using *libflush* on our test devices(Section: 1.6). Using this attack, we can identify the version of the shared library in use by the victim process. We can also identify which function is used by the victim process. In our sample victim application, a look-up function is performed in a look-up table using a shared library. Our attacker's threat model is based on the attack pre-conditions as mentioned in (Section- 4.2.1). As one can see from the results (Table: 4.1) that it is very easy to identify which address is being used in order to perform a lookup. Not only this, one can identify the location of the table within the shared library using this attack. This is an important result as various system applications like Keyboard etc perform a lookup in a large table. Important steps invloved in the attack are as follows :

- Identify the mapping of the shared library in the victim process's address space. This can be done by using */proc/pid/maps*. We can identify which all shared

libraries are used by the victim process using *ldd <name-of-the-executable>*

- Synchronize the spy process with respect to the victim process using proper mechanisms (file-locks, semaphores, signals etc).
- Calculate the cache-hits or cache-misses as per a suitable threshold.
- Run the EVICT+RELOAD attack using a spy application in parallel(on the same core or on the separate core) to the victim application to identify the addresses which have the largest number of cache hits (this will be the look-up index in the table)

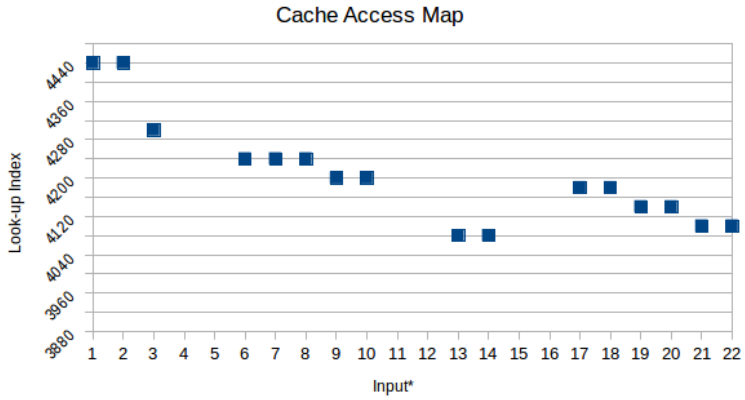


Figure 4.4: Cache Access Map of a process performing Table-lookups

*Note: *Input is XORed with a key* 245 *and then used to perform a lookup and is shown in Table: 4.1*

Lookup-index	Input*
0x4080	246,254
0x40c0	230,238
0x4100	214 ,222
0x4140	198 ,206
0x4180	182, 190
0x41c0	166 ,174
0x4200	128 ,134 ,142, 150 ,158
0x4240	128 ,134 ,142
0x4300	89
0x43c0	34
0x4440	1 ,13

Table 4.1: Cache Memory Access table

build

4.4.2 Cache Attack on AES Sbox

The very fast execution of AES seems to require hardware assistance in order to switch in between spy process and crypto/victim process. This heavily depends on underlying OS, CPU type and frequency. The main objective is to ensure that the victim process runs only for a small amount of time between any two runs of the spy thread. As suggested by Neve et.al [56], using `sleep` one may accomplish such a scenario. It is based on the fact that OS allows a process to control when it yields the CPU to another process without waiting until the end of the quantum. Thus, OS will reschedule the remaining quantum part to the victim process which will be able to execute some instructions and after which the OS will quickly resume the execution of the spy process. As per Neve et al. [56] the final strategy could be broken into following steps:

```
Spy : Continuously monitors the cache utilization of the parallel victim
thread may or may not be on the same core
Victim : Runs for very small time in between two runs of crypto
```

4.4.2.1 Cache Attack on AES: Pre-Conditions

For AES access-driven cache timing attack, in order to simplify descriptions and analysis of attack we will start with the following assumptions [1] based on the work of Zhao et.al [76] :

- The attacker uses uniformly distributed plaintexts.
- The attacker has access to an accurate and high performance timing mechanism for example cycle counter on ARM, *perf* or *monotonic clock*.
- The attacker should be able to operate synchronously with the victim process.
- Only once device needs to be rooted to upload kernel module to access performance counters (if they are used).
- The attacker knows the cipher text.
- The time to access data in the cache (a cache hit) and time to access data not present in the cache (a cache miss) always differs by Δ, where $(\Delta > 0)$.

4.4.2.2 Exploitation of Vulnerable Sbox Implementation of AES

As we are aware of the fact that first round of AES [30] uses the full key to perform a XOR operation between the input and the original key. The output of this XOR operation is used to perform look-up in the sbox table.

$$s_i = sbox[k_i \oplus p_i]$$

[1] For the sake of reproducibility, we have explained the most generic attack steps and assumptions in the current and the following sections

Sbox consists of only 256 elements (one byte each) which will fit into 4 cache lines on both our x-86 and our test devices (Section: 1.6). In each round, for 16 times this look up operation is performed. Hence 160 times sbox is accessed in one AES encryption but we will be able to see the effect on only 4 cache lines instead of 160 addresses (not all are different). It is because the cache operates on cache line level, not on the individual address level [57]. Hence, the probability of not accessing a cache line for b accesses in the first round of AES would be p(b):

$$p(b) = (1 - \tfrac{1}{4})^b$$

Since 16 accesses are performed in each round, then the probability would be as follows:

$$p(16) = (3/4)^{16} = .0100$$

As discussed before, our resolution is limited to one cache line, we cannot detect these 160 accesses independently. Cumulatively, our attack granularity is significantly reduced. In our vulnerable implementation, we have used these sbox lookups to perform a further look-up in a larger dummy table which is 4KB in size with each element of size 4 bytes. In this case, there will be 16 cache lines put to use [57], instead of 4 in our previous case and attack granularity enhances as shown below.

$$p(16) = (15/16)^{16} = .3560$$

Using EVICT+RELOAD attack on this vulnerable AES implementation as shown in Figure: 4.5, we are able to extract the full key.

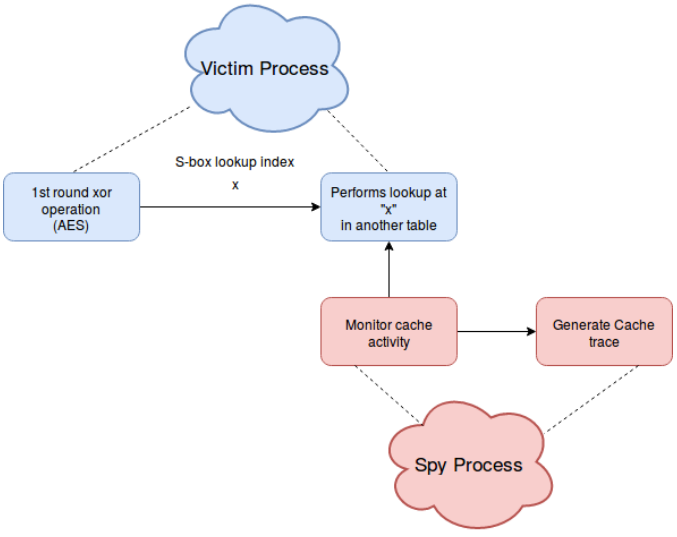


Figure 4.5: AES sbox attack strategy

However, the number of cache traces required increased significantly as the number of recovered key bytes was increased (shown in Figure:4.6). The possible reason for such an

increase in the amount of cache traces (Figure: 4.11) could be attributed to the resolution of attack which reduced the amount of useful information. Also, there is a noise in our measurements as we are operating in an environment where several other processes are executing simultaneously and might have an impact on the cache state.

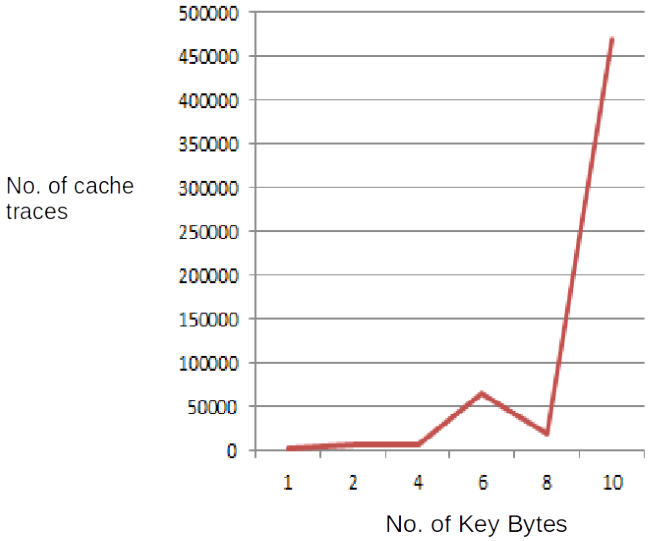


Figure 4.6: Number of bytes vs Number of cache traces

4.4.3 Cache Attack on AES T-tables

Our access-driven cache attack on AES is inspired by the works of Osvik et al. [58], Neve et al. [57], Spreitzer et al. [64] and Bonneau et al. [23]. In order to perform the attack on both x86 and ARM-v7 architectures we use the same attack strategy but different instructions for timing and eviction as mentioned in the Table: 4.2. In this

Architecture	Timing Mechanism	Eviction/Flush
x86	*rtdsc*	*clflush*
arm-v7	*libflush_reload_and_evict*	*libflush_evict*

Table 4.2: Cache attack enablers on X86 and ARM

attack neither do we have any information regarding when a T-table is accessed during an AES round nor do we get any information about the order of accesses within one measurement. We cannot identify any distinction between AES rounds. The perquisite is to map the shared library (*libcrypto.so*) used by victim crypto process into attacker's address space. We implemented cache attacks on both x-86 and our ARM based test devices (Section: 1.6). The threat model is shown in Figure: 4.7. We have worked

on two different attack scenarios while attacking OpenSSL T-table AES implementation (versions are 0.9.7a and 1.0.1g) and they are as follows:

- Cache template attack on the first round of AES
- Cache attack (EVICT+RELOAD) and (PRIME+PROBE) on the last round of AES

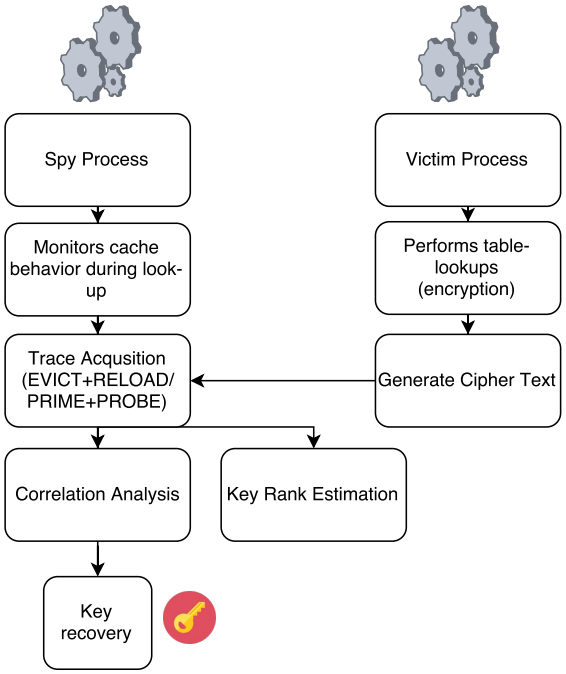


Figure 4.7: Threat Model

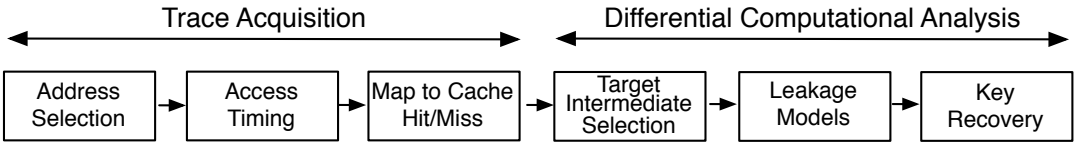


Figure 4.8: Attack Steps

4.4.3.1 Cache Template Attack on AES first-round

The first round attack is performed on sbox out of 1^{st} round of AES(on both versions 0.9.7a and 1.0.1g) and below-mentioned strategy is followed. In the first round of AES following operation takes place for i=1,..,16:

$$x_i = p_i \oplus k_i \tag{4.2}$$

build

Here x_i is the state byte which is used to perform T-table lookup for the next-round and p_i is a plaintext byte and k_i is a key byte. Hence, if we know the plaintext, then we can guess the upper 4 bits of the address being accessed (explained in Profiling Phase). The result would then be:

$$k_i = p_i \oplus x_i \tag{4.3}$$

The template attack consists of the following phases:

- **Profiling phase:** We generate a cache profile w.r.t a known key and a known plaintext. We perform this attack on one known key byte and one plain-text byte at a time. Also, p_0 is kept fixed and the rest of the 15 bytes are sufficiently randomized for n number of encryptions and then p_0 is incremented to another value from the set $\langle 16, 32, .., 255 \rangle$. This process is repeated for all the 16 bytes of the plaintexts if we want to recover 4 MSBs of the remaining key bytes as well. In total, there are 16*256 combinations for each plain-text bytes. The profiling phase consists of the following steps:
  - In the first step, we run AES encryption in order to load the T-tables in the cache. This step can be skipped if we want to start with an empty or uninitialized cache. The result of the attack remains unchanged.
  - In the next step, we perform eviction of a cache line followed by AES encryption. AES encryptions are performed on randomized plaintexts with p_0 being fixed in the first step of the attack.
  - In the final step, we access the evicted address (cache line) again and measure the time taken to access that address. If the time taken is above a certain threshold then there was a cache miss otherwise a cache hit. Step (2) and Step (3) are repeated for n(>300) number of encryptions. On the basis of such information a cache profile of cache hits and cache misses with respect to each cache line is generated. Such profile can be used to identify the 4 most significant bits of the key byte being used. For example, if the key byte, k_0 is 0x5f which will be a part of (0x50,...,ox5f) set and plain text p_0 is 0, then a lookup will be performed on the cache line containing all these 16 addresses of T_0. The vertical columns in the Figure: 4.9 in the cache trace define the cache lines occupied by T-tables.

  Figure- 4.9 depicts the image of profiling phase against $k_0 = 0x00$ and plaintext, $p_0 = (0, 240)$. This process can be repeated to recover all the key bytes.

- **Attack Phase:** In this phase, we perform encryption of known/chosen plaintext with an unknown key using the steps mentioned in the profile phase. Since the key is unknown, we can compare the newly generated cache profile with the profile phase template. In the current scenario k_0 is 0x52 and $p_0 = 0$. An XOR operation would result in a lookup index in T_0 which is:

$$look\_up\_index = 0x52 \oplus 0x00 \rightarrow 0x52$$

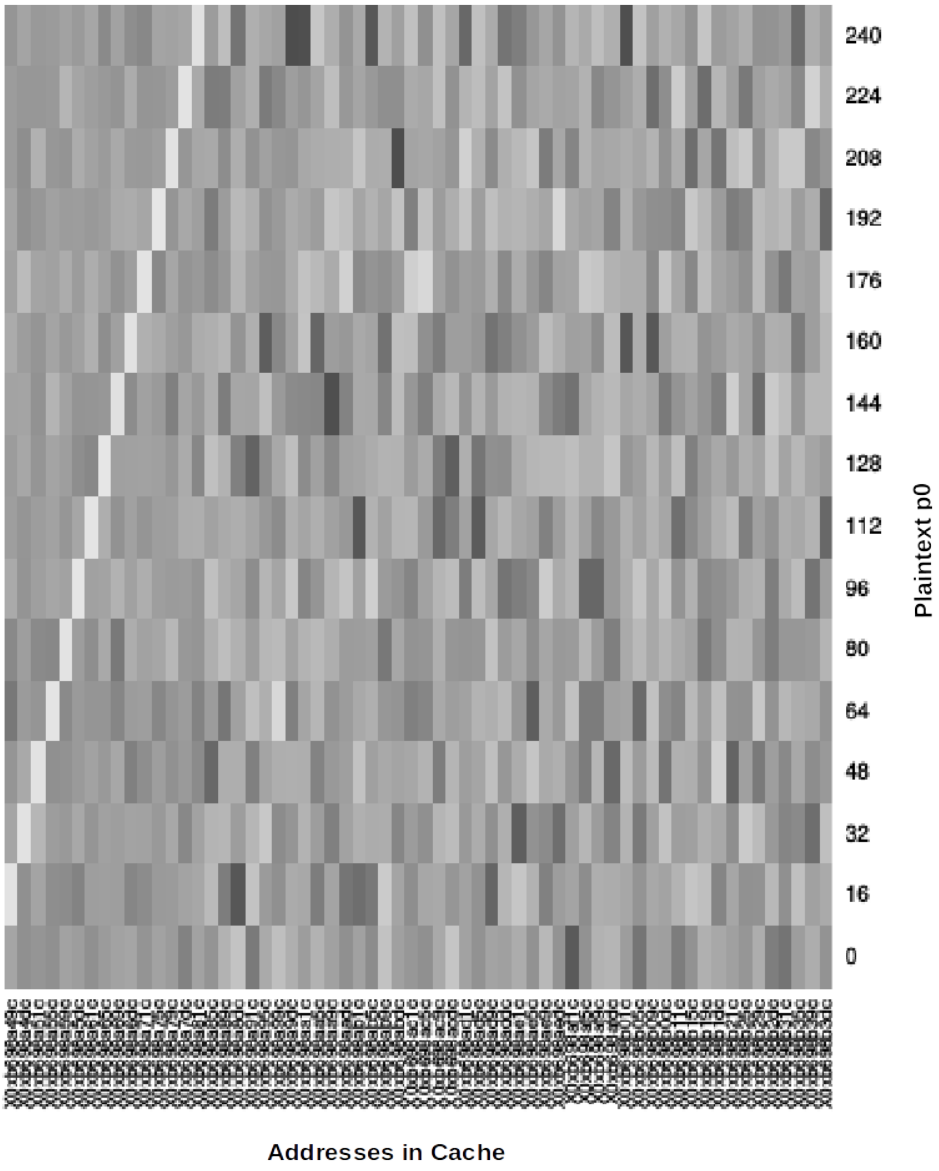


Figure 4.9: Nexus 5: AES profile trace for k_0=0x00

build

and hence there is a cache hit (brighter region) in the fifth cache line. Only 4 most significant bits can be identified accurately which result in **5** as we are operating at cache line level where 16 different lookups in the same cache line would result in a similar pattern. Hence for all key candidates $k_0 = (0x50, ..0x5F)$, there would be similar cache patterns. The final outcome of this attack on (OpenSSL 1.0.1g) Nexus 5 is that the key entropy is reduced from 128 bits to 64 bits. Similar results are achieved on Samsung Galaxy S4.

4.4.3.2 Cache attack on AES last-round

Most of the attacks on AES are based on first and second rounds of AES encryption [58][57][64][23]. Since first round is disturbed by the access of other rounds, hence we chose to attack the last round. Neve et al. [56] has proposed elimination and non-elimination methods to use ciphertext in order to recover the entire key while attacking last round of AES. Our attack is different than the strategy proposed by Neve et al. [56] as we use *Correlation Analysis* on our cache traces, however, we do share some nuances of their work.

We have already discussed that in OpenSSL 0.9.7a AES implementation, T_4 is used only in the last round and depicts the end of encryption. T_4 is accessed 16 times during the last round, hence the probability that a cache line in T_4 is not accessed in m encryptions is p(m) denoted by:

$$p(m) = (\tfrac{15}{16})^{16} \rightarrow .356$$

Our EVICT+RELOAD attack works on both T-table AES implementations (OpenSSL 0.9.7a and OpenSSL 1.0.1g). We start with mapping the shared library under attack (libcrypto.so) in attacker's address space. In this instance of the attack, the encryption and spying execute as two forked processes. This scenario is less realistic than the context where victim and spy are different processes. However, we discuss the results of PRIME+PROBE which works on two different processes (steps are the same as discussed in Section- 4.3.2). Our EVICT+RELOAD attack consists of three phases :

- **Profiling Phase:** In this phase, we attempt to find out the location of T-tables in **libcrypto.so** using any linux utility like *objdump*,*gdb* etc. This is done to reduce the number of addresses used for probing in the actual attack. In this attack scenario we may also disable the ASLR to find the exact location of T-tables. However, we noticed that even if we do not disable ASLR (explained in Section- (4.4.2.1)), tables are slightly misaligned and that does not really affect our attack intensity. The profiling phase provides with the information about the location of T-tables in the cache memory i.e. which cache sets are occupied by the T-table elements.
- **Exploitation Phase:** Once the location of t-tables is identified, we perform a warm up encryption on plaintext to load the T-tables in memory (this step may be skipped). We will perform $n > 8000$ number of encryption for each cache set

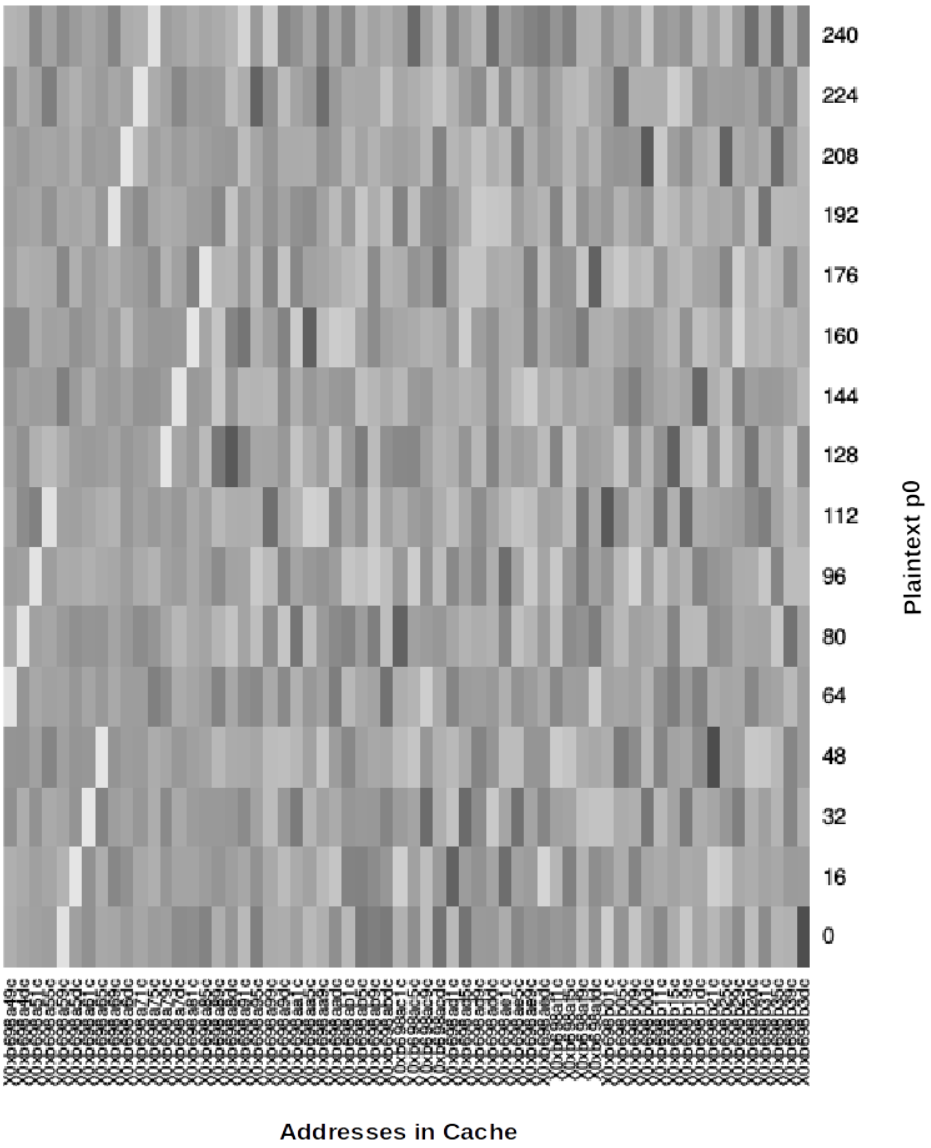


Figure 4.10: Nexus 5: AES cache trace for $k_0 = 0x51$

build

occupied by the T-tables. For demonstration, we use OpenSSL 1.0.1g (32 byte version) on Android (attack should be scalable to other versions as well), the total space occupied by $T_1 - T_4$ tables is 4KB. Hence, the number of cache sets occupied by T-tables is 64. Then, we evict a specific cache set from the T-tables and perform encryption again. If the evicted cache set is used by the encryption, then it will be fetched again from the memory otherwise not. In the last and final step we access the evicted address again to check if it is loaded by the encryption step. We will store the the plaintext (not necessarily required as we are attacking sbox-in of 10^th round), ciphertext and corresponding CPU cycles w.r.t each cache set in a structure called cache_trace 4.11.

Attack steps

```
1  start_addr = start address of T-tables
2  end_addr = end address of T-tables
3  struct cache_trace
4  for i =start_addr ;i <end_addr ;i+=64
5     aes_encryption()
6     for j =0; j< NUMBER_OF_ENCRYPTIONS ; j++
7       eviction of ith set
8       aes_encryption()
9       calculate time to access evicted ith set
10          cache_trace.plaintext = plaintext
11          cache_trace.ciphertext = ciphertext
12          cache_trace.timings = timings
13    end for
14 end for
```

- **Correlation Phase:** In the correlation phase, we perform the correlation between the intermediate value and every bit of the cache address being accessed or not. In order to perform correlation, we convert our cache traces in binary traces by using a suitable threshold. We use different leakage models like Bit, ID, and Zero Value models to hypothetically determine the intermediate value. Another most important technique which is utilized to identify how many traces are sufficient in order to recover the full key is *Key Rank Estimation.*

Input Output Addresses

c0f51069e5705e8171bc75d	2ed1cdc628048bcb7c4087a1eb8597d1	93	264	264	90	100	96	252	264	267	87	264	90	75
e2cd0730e81d256c32772d3	4f08e43ffefbdb78074087622f04cc41	78	261	267	228	276	316	96	72	228	90	267	249	87
67cafd4d3a2dceace943870!	61918de57ea52074cfa8ddd5ea334d7e	93	225	87	264	100	256	96	90	87	87	72	243	87
cc190cb4373120693a84e9a4(	3c9f70e3953606f5c2ac5a704ef0459c4	93	90	87	264	100	248	252	72	90	87	90	267	90
1f827459af430598868c6d73	f5dc41e47baec2134273155ce80ed461	87	240	264	90	100	100	100	225	267	87	264	75	231
994e0bd07f2b392779d4753	d5d75f6284afba4aa82cecc3170c553	93	90	267	90	96	252	288	87	264	87	90	228	90
abd5a15a18a6f39f3360120f	21df7334829b6ad4b016c786ba171b71	93	240	90	264	252	100	252	90	267	87	267	90	90
a3204a234b644ac558bfffe80	ffb2b4f6cb9e80ddcef3e640e39963171	93	243	267	87	304	248	96	72	90	243	72	252	90
3569ed4d0fe0ec4241ff0021	61b0d25f25b14e9992338b853fac87c5	93	87	267	90	348	844	96	267	75	252	87	87	87
35e61a816a6546c224444420(	ecd96207ca615b37b963d5a3c3420ff1	90	246	228	267	96	280	100	75	90	249	87	240	87
738d26836d13c5ae12c6d06l	b8476dd93dfe0101326b9a48323a0b77	90	264	855	90	272	100	100	87	90	225	225	264	264
517892bbdd87d011cc00ec	3b9b824b56d92401791632072178 5c4e	93	87	90	87	96	96	284	264	270	90	87	264	90
3ac0c1a8d386569e94264b2(	41932010d523a4e05a31d1db1078604e	87	90	267	267	252	96	252	264	264	225	747	90	90
20cb9ffda31cffbfd1c0d80a7(	03ff13aabdf77e29fc0ef1ca5a0561e7	93	90	603	264	100	100	264	252	87	228	477	321	87
d9a690dac2cf291791adca1ac	cae34d2ca15f9943f87dae7c04e469a	93	90	243	87	100	252	284	333	87	87	90	243	267

Figure 4.11: Cache Traces

build

5 Results and Analysis

In this chapter, we discuss and analyze the results of various cache attack strategies on different architectures, namely: ARM and x86. The focus of our results and analysis is on our test devices (Section: 1.6). We run a pilot attack on x86 first, to reckon the feasibility of our attack and then we escalate it to our test devices 1.6.

5.1 Cache Correlation Analysis (CCA) and Leakage Models

Correlation coefficient provides an extremely efficient way to determine linear relationships between data. In our case, we aim to find correlation between the hypothetical intermediate values for every possible key guess to binary cache trace (which contains 1 for a cache miss and 0 for a cache hit on a bunch of addresses) based on certain leakage models. It is to be mentioned that only 4 MSBs of the 16 attacked addresses are important as cache traces operate on cache line level (containing 16 elements each) not on individual address level. We attacked sbox-out of 1^{st} round of AES encryption on Nexus 5 to demonstrate the applicability of cache template attacks. We use our attack strategy along with various leakage models [65] (they are used for power-side channel analysis) to attack AES last round on our test devices(1.6). In this section we will discuss these leakage models and their usability w.r.t our attacks scenarios. We use Inspector (an advanced tool for side-channel analysis) [10] to perform correlation analysis which we term as *Cache Correlation Analysis* (CCA). Any other script or tool which is capable of performing CA can be used, hence using INSPECTOR is not part of the requirements. Thus this attack can be completely performed in an online mode, there is no requirement to analyse the cache traces separately on another platform.

The non-linear behavior of sbox [65] is key to this analysis. A one bit difference at an sbox input leads to a difference of several bits at the output. Hence, even if a key hypothesis is only wrong in one bit, the output in sbox will be different in several bits. Therfore, while attacking the output of sbox the correlation for all wrong key hypothesis is significantly smaller than the correlation for the correct one. Hence, we mount our correlation analysis on the intermediate results that occur after the sboxes (sbox out) in the first round and before the sboxes in (sbox in) last round.

- **Using ID Leakage Model**

  ```
  ID + 0--------------------N
  ```

  ID leakage model operates on byte level and is suitable for addresses which are close to the either extremes (beginning(0) and ending(N)) of the table. So an address

(an index used for lookup) which is either close to 0 or close to N, there will always be a very high correlation. For e.g in the case of address 0 there will be a high correlation when the intermediate value is 0 and if it's higher, the result would be opposite. This is a linear dependency (0: yes, >0: No). The inverse holds true for address N. For all addresses in between, this dependency slowly decreases towards the middle and goes to the negative extreme for N. The reason can be attributed to the fact that the impact of entire byte address correlating with intermediate value is taken into consideration instead of the 4MSBs. Additionally, in such a case neither low values nor high values increase the chance of a hit –> no linear relationship –> no peak. The effect is plotted in figure:5.1:

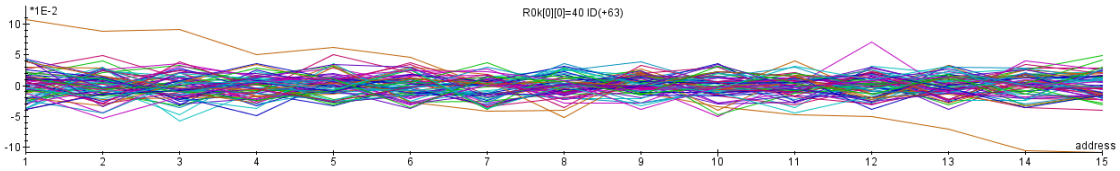


Figure 5.1: ID model for binary cache trace

- **Using Bit Leakage Model**

  `BitX + 0--------------------N`

  This model depicts a linear dependency in between the model and the addresses being used during encryption (as shown in sample cache traces) and therefore it is detectable with correlation. It is nice for White Box Cryptosystems since we can circumvent several encoding schemes. This model gives rise to interesting patterns as we take into consideration only a part of the index (address) in the correlation trace.

  In order to simplify the assumption lets take into account how 16 cache line addresses would be represented in binary Address0: 0000
  Address1: 0001
  Address2: 0010
  Address3: 0011
  Address4: 0100
  Address5: 0101
  Address6: 0110
  Address7: 0111
  Address8: 1000
  Address9: 1001
  Address10: 1010
  Address11: 1011
  Address12: 1100

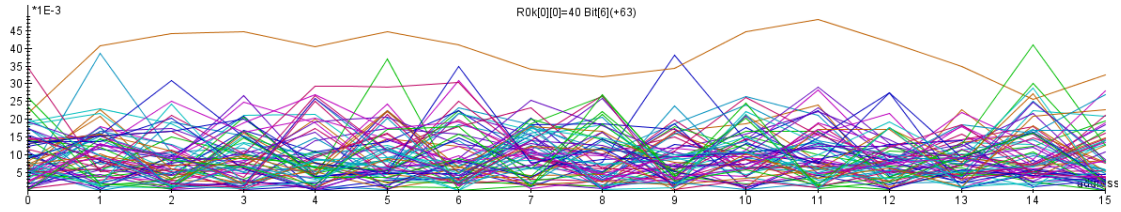
Address13: 1101
Address14: 1110
Address15: 1111

Now if we correlate the MSB of our intermediate (which is the index for this table of addresses) we get some interesting results. For example: having the MSB at 1 means that addresses 0-1 and addresses 4-5 and so-on so-forth cannot be used and having the MSB at 0 means that there is a 50% chance that addresses from 0 to 16 are alternatively used and one can spot a negative correlation at an odd number of addresses.
This effect is shown in (Figure:5.2b) for the 4 most significant bits (MSBs) of a 8 bit lookup table (sbox) on x86 architecture (Intel i5) based *DELL* machine . (Figure:5.2c) also, demonstrates the effect of 7^{th} MSB of the index value. **This information about the correlation in between the MSB of the address and the bits of intermediate value is of significant importance**. As now, we know for sure that which leakage model depicts our cache trace characteristics in the most accurate way. As bit model gives the bit level granularity (shown in Figure:5.2a) and depicts the correlation of each bit with the intermediate value, it is a suitable choice for cache correlation analysis in the later experiments.

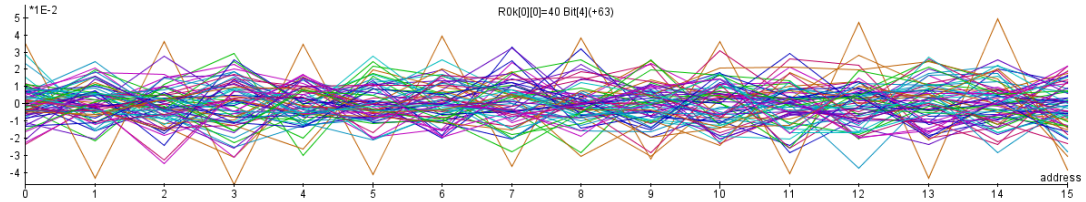
5.2 CCA on Different Processor Architectures

5.2.1 EVICT+RELOAD

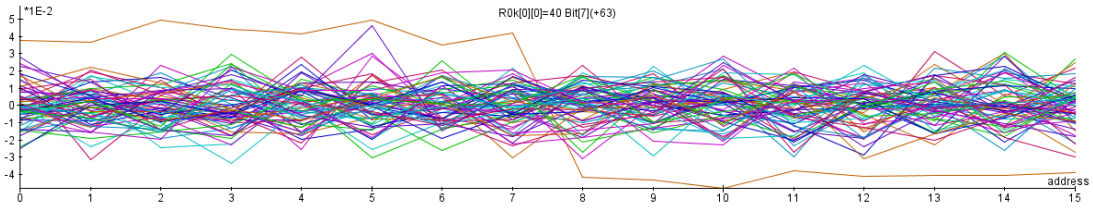
EVICT+RELOAD cache attacks are succesful in full key recovery on x86 architecture as expected and we are able to recover the full secret key from OpenSSL 0.9.7a AES implementation. Owing to the availability of accurate timer and unprivileged cache flush instruction, our task is much simplified. Figure:5.3b demonstrates the minimum-average-maximum number of cache traces required to recover the full key from a crypto process running on x86 (Dell Inspirion 15-5000 series). If we do not use any other statistical tool to our advantage, then the minimum number of encryptions performed are $\approx$ 1000 (Figure: 5.3a). We would like to mention here that we are making use of *sum of peaks* property of correlation analysis. This helps us increase the overall impact of the cache traces on the final results and thereby reduces the number of required traces for full key recovery. In the following results, we are taking into consideration only the information related to the 4MSBs of the addresses, as cache operates at line level and each line contains 16 elements. Therefore, only 4 significant bits of the addresses contain the useful information and the rest contain redundant information. However, for comparison purposes, we also investigate how results vary when information related to every single address is taken into consideration.
While analyzing different trace-sets (Figure:5.4 and Figure:5.5), we figured out that despite noise on the system and other environmental factors, we could recover the full key with only $\approx$ 900 traces. Before using Key Estimation Algorithm, we were making use of almost 9500 encryptions to recover the full key as we were unaware of the bounds. Figure:5.5 demonstrates the bounds using various trace-sets.



(a) x86: Bit Model for all bits of address



(b) x86: Bit Model for 4^{th} MSB of address



(c) x86: Bit Model for 7^{th} MSB of address

Figure 5.2: x86: Correlation results of Bit Leakage Model

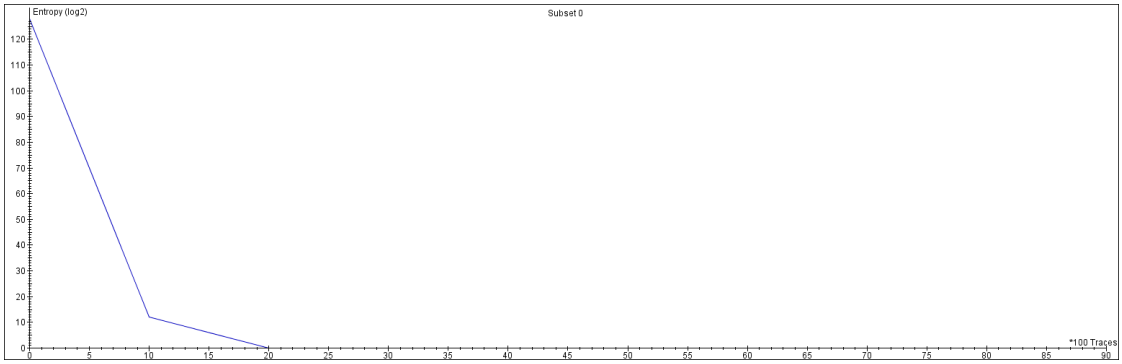
Number of Traces	Remaining Bit Entropy(with peaks)
900	0
1100-1500	9
1700	15

Table 5.1: Reduction of bit entropy with number of traces

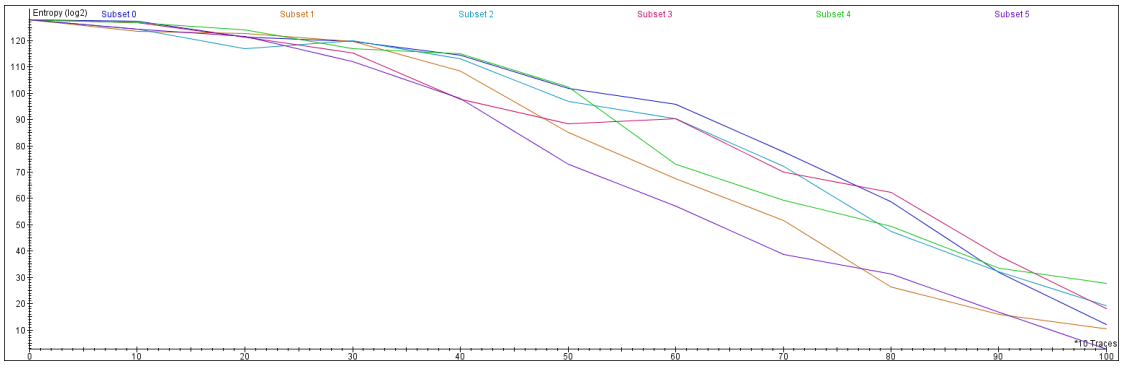
5.2.2 EVICT+RELOAD on ARM

The results of EVICT+RELOAD attack, on OpenSSL 1.0.1g AES implementation, on ARM architecture vary quite significantly w.r.t no. of traces. **The Binary Threshold applied to our cache traces is:180.**. Figure:5.6b demonstrates that even 10000 traces are not sufficient to recover the full key from a crypto process running on *Nexus 5* (using

build



(a) Key Rank Estimation Using 9500 encryptions



(b) MIN-AVG-MAX Key Rank Estimation Using Sum of the Peaks

Figure 5.3: x86: EVICT+RELOAD attack

Subset												
	128.00	127.51	121.24	120.15	114.62	101.55	96.13	77.84	59.45	34.17	13.85	4.81
	128.00	119.31	120.83	106.91	87.83	76.17	53.21	31.66	18.83	15.26	6.88	0.00
	128.00	127.18	126.47	114.63	115.14	99.63	75.08	68.26	49.29	47.04	32.60	8.81
	128.00	122.69	119.60	116.37	111.02	100.78	75.39	52.54	38.82	29.23	18.27	8.43
	128.00	125.79	116.98	116.81	111.42	89.75	90.26	78.08	50.84	17.27	5.86	4.32
	128.00	119.18	120.43	119.55	101.34	97.40	85.71	77.72	56.19	31.25	17.91	7.21
	128.00	123.68	119.77	113.15	105.08	81.10	65.17	41.55	9.48	1.00	1.00	0.00
	128.00	125.43	122.87	117.77	100.48	82.30	64.24	70.96	43.06	44.66	28.37	17.70

Key Entropy after every 1000 traces

Figure 5.4: Subsets and the reduction of key entropy

Key Estimation Module of Inspector). However, the bit entropy is reduced significantly and remaining bits can be recovered using brute force. We compare the results on the basis of number of traces in the following 4 figures (5.6a,5.6b,5.7,5.6d). As one can clearly

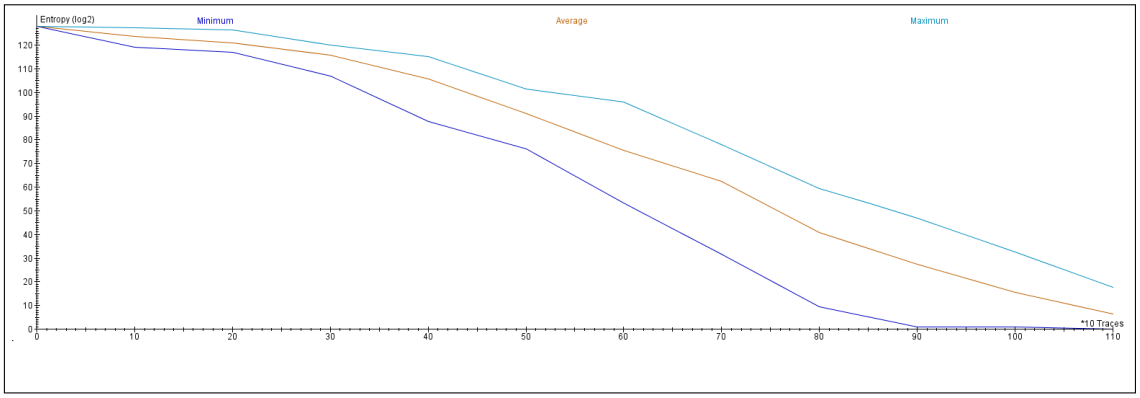


Figure 5.5: MIN-AVG-MAX Key Rank Estimation on 1100 traces

observe, more the number of traces, more the number of recovered bits.

$$Number\_of\_traces\, \alpha\, Number\_of\_recovered\_bits \tag{5.1}$$

There is an interesting effect which can be highlighted by using cache hit/miss ratio of every individual address instead of cache line addresses and then try to recover the full key. As one can see from Figure:5.8, that tables are a little misaligned and not every 16 address present in a cache line depict the same behavior of the line (due to misalignment). Figure: 5.8 and figure: 5.9 demonstrate the effect of key rank estimation and correlation analysis on our cache traces (post Binary Threshold Filtering).
Next, we demonstrate the results related to *Samsung Galaxy S4*. In the case of Samsung Galaxy S4, the minimum number of traces required to recover the full key is more than 8000 as can be seen from Figure:5.10a and Figure:5.10b. They collectively show the difference in results with or without sum of peaks while estimating the number of traces required to recover the full key. It is shown that the key bit entropy gets reduced to 48 bits, if we take into account individual peaks in correlation analysis while using only 8000 cache traces. On the other hand, using the *sum of the peaks* property, the key bit entropy gets reduced to 14.5 bits.

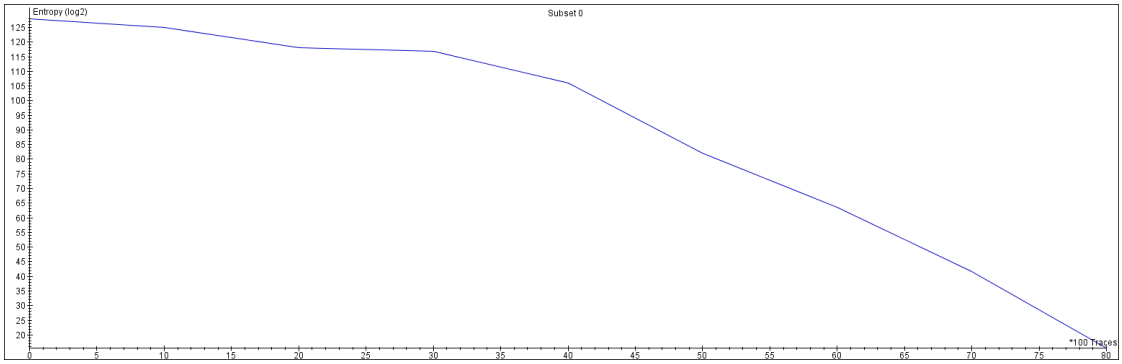
5.2.3 PRIME + PROBE

5.2.3.1 ARM

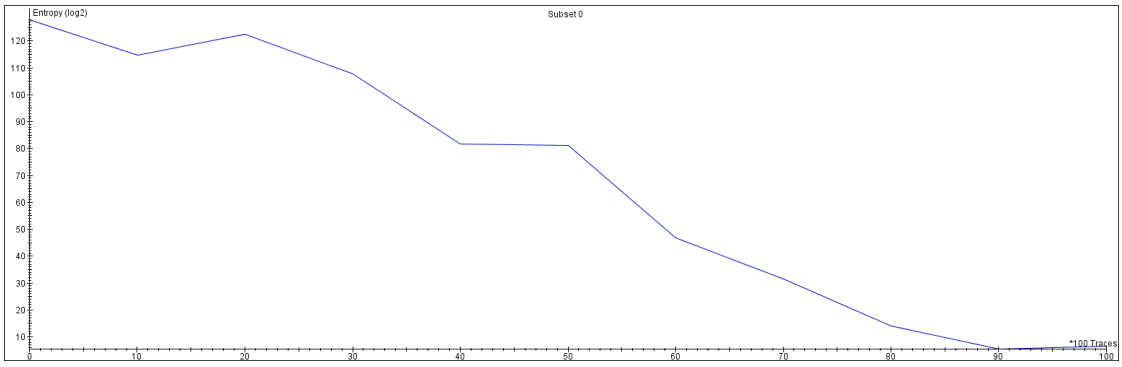
The key estimation results associated with the PRIME+PROBE technique differ from the EVICT+RELOAD because of the following reasons:

- Since there is no concept of dedicated shared memory involved in between two processes in PRIME+PROBE, there are more chances of false positives. Filtering of results on the basis of additional parameters is required.

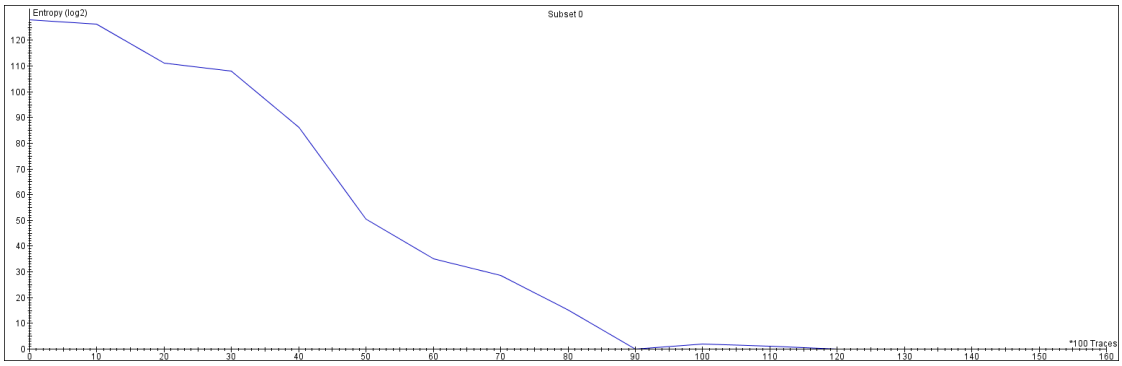
- Noise is added as some other processes can cause evictions and may hamper the results in the cache trace sets.



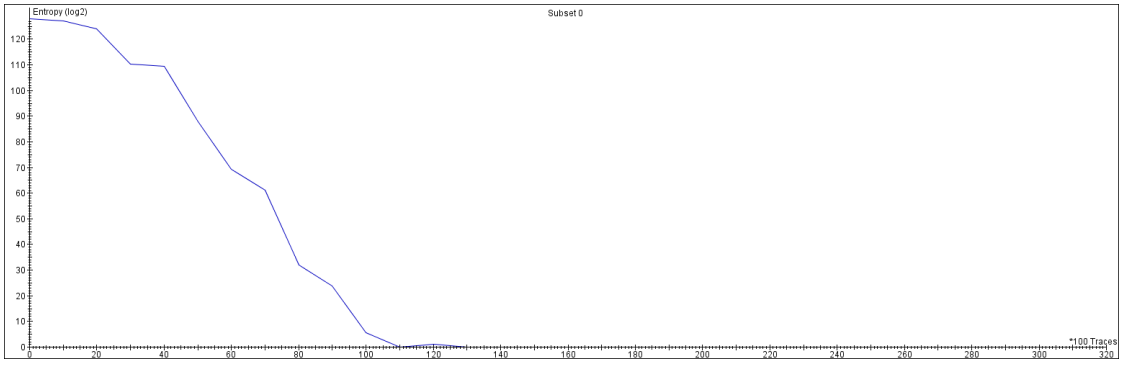
(a) Key estimation w.r.t 8000 cache traces



(b) Key estimation w.r.t 10000 cache traces



(c) Key estimation w.r.t 16000 cache traces



(d) Key estimation w.r.t 32000 cache traces

Figure 5.6: Nexus 5: EVICT+RELOAD results of key Estimation on varying number of cache traces

build

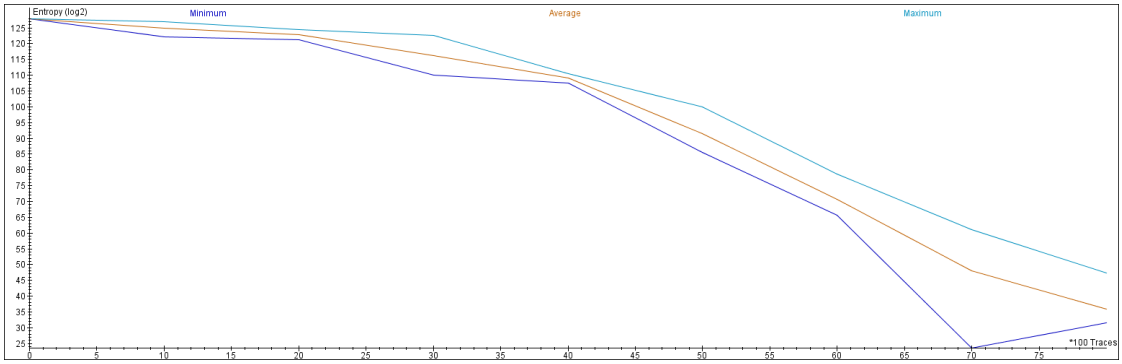


Figure 5.7: Nexus 5: MIN-AVG-MAX key rank estimation

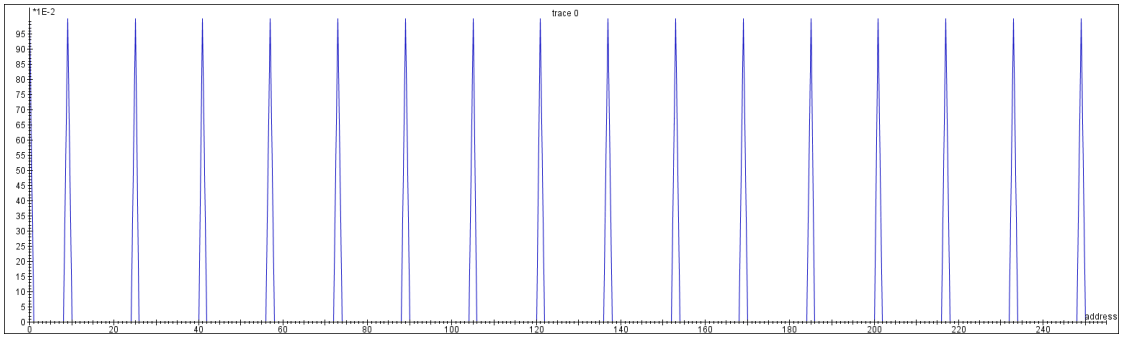


Figure 5.8: Nexus 5: 256 addresses depicted in Cache traces

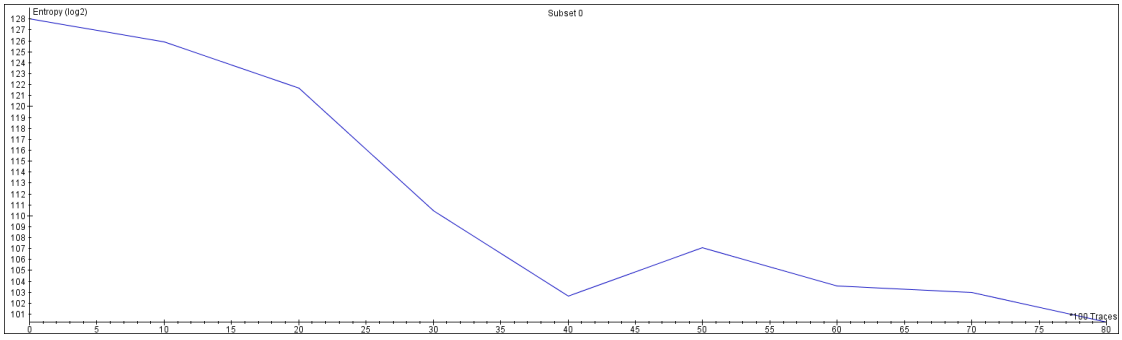
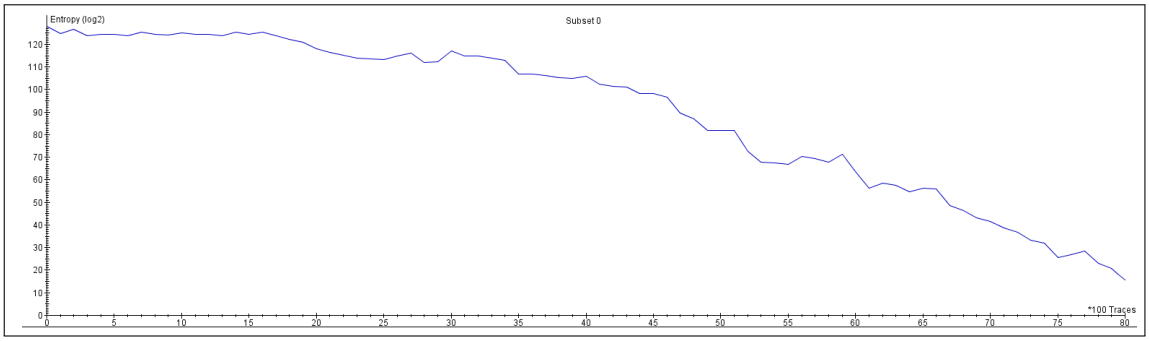


Figure 5.9: Nexus 5: Key Rank Estimation using 256 addresses

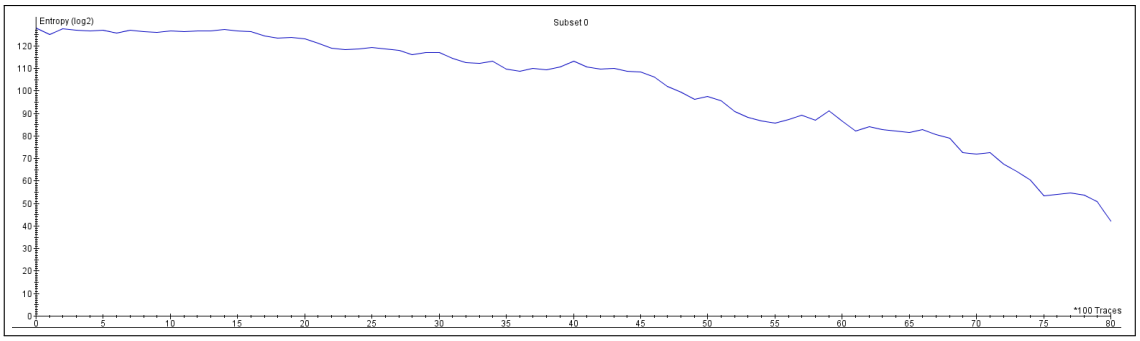
- Also, it requires more addresses to be monitored in comparison to EVICT+RELOAD. Hence, it is slightly more time-consuming.

We adopt the same number of traces as used for EVICT+RELOAD technique as a benchmark for PRIME+PROBE technique. We observe that initially we are not able to recover the full key on Nexus 5 or Samsung Galaxy S4 with 8000 traces as other processes were running on the smartphones, however, key entropy reduces to a certain extent as shown in Figure:5.11. Later, we run only our crypto and victim applications on our test devices (Section: 1.6) and the results are shown in Figure: 5.12.

build



(a) AES sbox leakage using sum of peaks of 8000 traces



(b) AES sbox leakage using individual peaks of 8000 traces

Figure 5.10: Samsung Galaxy S4: EVICT+RELOAD results

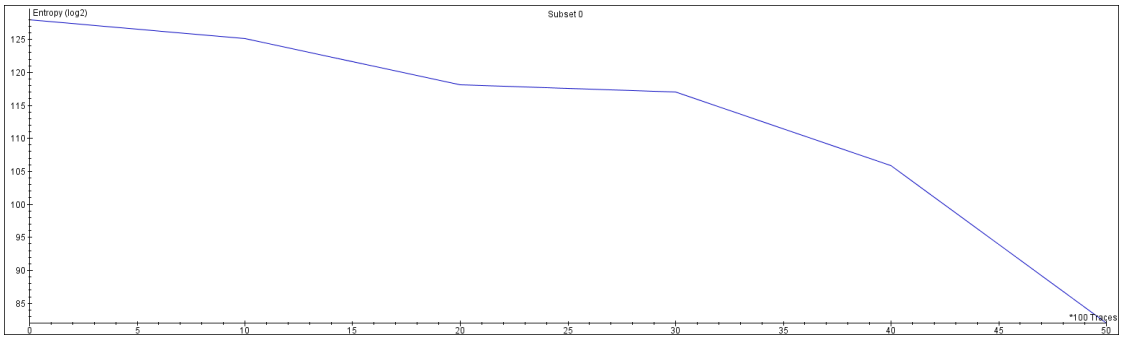


Figure 5.11: Nexus 5: Prime+Probe Results with other processes running

5.2.4 White-Box Cryptosystems

The most important goal of attacking a white-box [12] is to extract the key of the cryptographic algorithm. The first and most important step of attacking a white-box is to find an attackable white-box [12]. It may look like a trivial step but it is a necessary and important step. Real-world white-boxes [12] are usually integrated into bigger applications and attacking those would be a challenge as they must be well protected against Dynamic Binary Execution (DBI). While analyzing the white-box with one trace we may identify which encryption is used. Once we are able to identify where the white-box is, then we

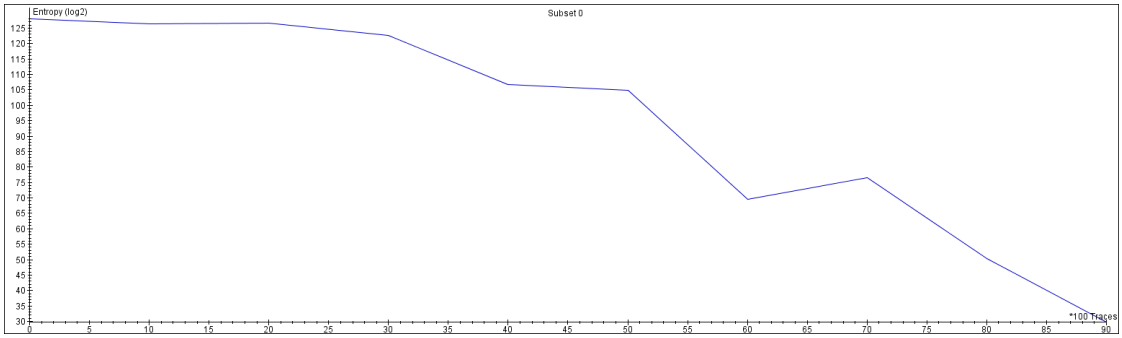


Figure 5.12: Nexus 5: Prime+Probe Results with only spy and victim process in runnable mode

may finally limit the memory addresses to the minimum which is required to recover the key. The next step is to create cache traces which focuses on the white-box execution.

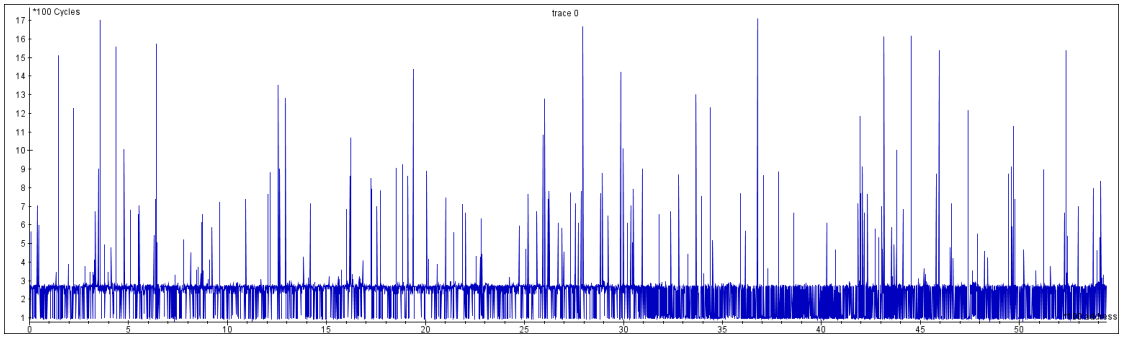
We follow all the above mentioned steps and then aim to mount a side-channel attack on the crafted traces. Finally, we want to find co-relation between the known memory traces and the ideal data. It should be noted that whether a side-channel attack would work or not is very difficult before trying to attack it. For public white-boxes, it possible to recover keys without any reverse-engineering or further knowledge of white-box [33][12][24]. We tried our attack on a custom white-box implementation (proposed by Chow et al). However, we could not reduce the entropy beyond 120 bits while using only 2000 traces as shown in Figure:(5.13a),(5.13b and (5.13c). Our results also suggest that there is linear relationship between the number of traces and the key entropy. If we increase the number of traces then we should be able to recover the complete secret key. In our work, we confined ourselves to 2000 traces as the trace acquisition alone took more than 30 minutes.
It can be attributed to the following reasons:

- The tables used in this white-box implementation are large in number and are very small. They could fit into one cache line. This hampers our attack results as our granularity of monitoring is limited to one cache line.

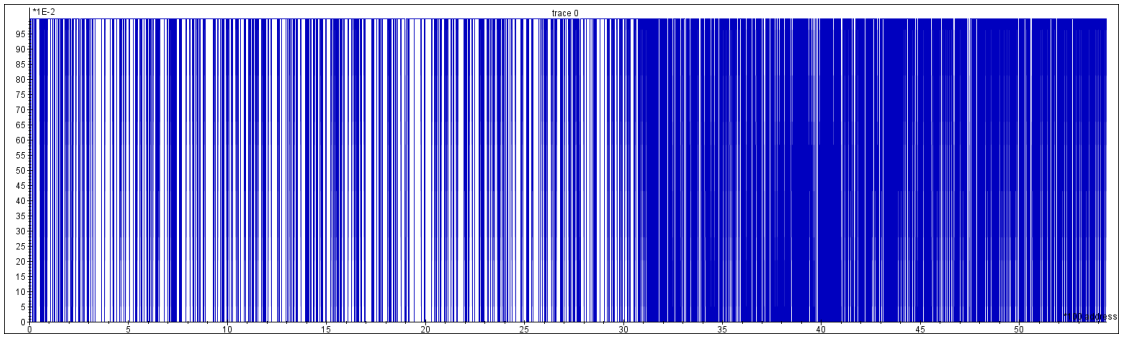
- Secondly, we cannot perform a profile phase as these tables are randomly created at every execution.

- During the generation of the tables in this white-box, it has been ensured that they perfectly align to a cache-line. Hence, as one can easily understand that this white-box was particularly created while keeping cache-attacks in mind.

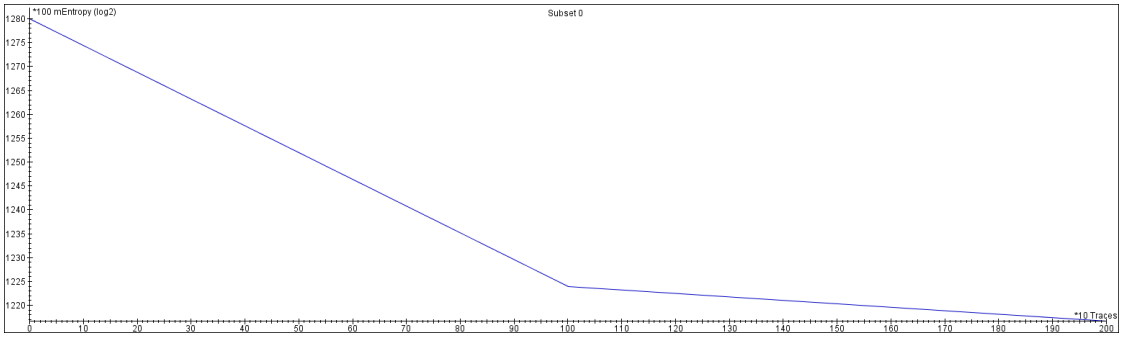
We tried to attack several academic white-boxes but they were immune to our attack owing to one or another above mentioned reasons. It should be noted that input/output encodings do not have any impact on our results. This finding could assist while attacking future white-box implementations where encodings do annoy the attackers a lot.



(a) Cache trace-set for access time w.r.t addresses used by white-box



(b) Binary Threshold applied to cache trace-set



(c) Key Rank Estimation on the newly generated trace-set

5.3 Impact Analysis of Attack Assumptions

We aim to analyze the impact of various attack assumptions on our results.The assumptions would be analyzed with respect to the success rate (success rate refers to the percentage of successful recovery the secret key during an attack) Following are the results w.r.t various attack parameters:

- **Impact of prefetcher:** While attacking an artificial application when we used de Bruijn cycle, we could avoid the effects of hardware prefetching. On the other hand, when we accessed the array elements in a progression, it triggered the hardware prefetcher as expected.

- **Impact of ASLR:** In case, we do not disable ASLR then the AES t-tables are

slightly misaligned. It does not completely affect our results although adds a bit of noise. On the contrary, in some cases it can also be beneficial as one can directly get the key byte as shown by Spreitzer et al. [64].

- **Impact of operating System:** The Linux offerings like *shared memory*, *mmap* and *pagemap* are exploited in our cache attack. Android is based on Linux and as discussed in Chapter-1, Android follows principles of sand-boxing etc to protect its application but the mysterious area of shared hardware is not paid attention to. Privileged access to cycle counter and cache flush instructions do make our task a little bit difficult but cache coherency came to our rescue. We could deploy the cross core cache attacks on our test devices 1.6 using cache coherency offered by AMBA ACCI.

- **Impact of cache state before the attack:** Our attack does not work on the cache misses but on the cache hits unlike previous works [58][26][77][40]. There can be three different scenarios in which our cache attacks would perform differently as stated below :

  - **Empty initial state:** This is the most favored state [18] for cache attacks as it provides with cold cache misses which were generally the basis for previous attacks on x86. We used this cache state to test our attacks. However, our attack works even without an empty initial state.
  - **Forged initial state:** In this sceanrio [18], the attacker should be able to control the initial state of cache as per his/her requirement. This is suggested to manipulate the number of cold start misses. It is equivalent to empty initial state in a way as it also involves flushing of cache followed by some fake encryptions. However, there is a difference as it uses conflict misses instead of cold misses to gain information about the significant key bytes.
  - **Loaded initial state:** As the name suggests [18], if the tables are already loaded in the memory then that state is called loaded initial state. We utilized this state as well for our attacks and it reduced the number of traces required to recover the complete key.

- **Number of cache traces/encryptions:** The relationship between the number of traces or encryptions required and the success rate of our attack are linearly dependant. If we have more traces, that amounts to more information and eventually, the chances to recover the complete key increases as well.

- **Impact of the privileged and unprivileged mode:** Industry has always argued that if an attacker has an access into victim's execution space then there is no point in carrying out a more complicated low level microarchitectural attack. To answer this, we made our attack work in both privileged and unprivileged modes. However, in a privileged mode we have access to cycle counter, consequently, the success rate increases and the number of encryptions required decreases. In unprivileged mode, we made use of timing mechanisms like *perf* and *monotonic clock*. The results were still promising, however number of required traces increased with the change in the choice of the timing mechanism.

- **Impact of eviction strategy:** It is the most crucial aspect of the attack as an eviction strategy with 100% eviction rate and lower execution time is essential. Using Cache Eviction Strategy Evaluator [53] our task became fairly easy and we could come up with an efficient strategy. In this study, we did not try the eviction strategies suggested by [58][79] as they were time-consuming.
- **Impact of the type of attack:** Prime+Probe is a much realistic attack but we could not completely recover the key using Correlation Analysis, given the same number of traces used for EVICT+RELOAD. On the other hand, EVICT+RELOAD in various attack scenarios results in a successful key recovery as discussed in previous sections.

build

6 Conclusion

6.1 Conclusion

We present a novel evaluation strategy for cache side channel attacks on Android smartphones(1.6). Our study discusses the applicability of cache attacks (PRIME+PROBE, EVICT+RELOAD) on a victim application making use of native code on our test devices (1.6). It emphasizes the effect of the cache intial state, timing probe, compiler, access settings, operating system etc on the generated cache traces. Further, we discuss the number of key bytes recovered by analyzing the cache behavior. Our experiments are not entirely dependant on the above-mentioned specifications and hence can be reproduced with much ease. We perform cache attacks on OpenSSL 1.0.1g and OpenSSL 0.9.7a AES implementation on our test devices 1.6 using a state-of-the-art attack strategy based on the works of Spreitzer et al. [64] and Tromer et al. [58]. In the previous works [58][20][38] picture analysis was fundamentally used to identify the key candidates. Correlation Analysis was used in timing attacks but not in access driven cache attacks on ARM. **Our results fulfill the main objective of our research which is the successful recovery of the complete secret key of AES (a crypto algorithm) on ARM devices with the assistance of statistical techniques like *Correlation Analysis and Key Rank Estimation*.** Using the proposed method, we quantitatively evaluate the transition with the increase of the number of plaintexts. We also discuss the performance of our attack in varying attack scenarios and the effect of the cache state on our attack performance is also analyzed. Finally, we demonstrate the cache attack on a vulnerable implementation of AES (resembling a white-box implementation) and then test its applicability on a custom white box implementation of AES.

Future work related to the investigation of cache attacks on white-box cryptosystems seems promising. We believe that incorporation of statistical analysis with access-driven cache attack opens door to a major research in the field of white-box cryptosystems. Furthermore, launching these cache attacks without any privilege escalation or assistance from performance counters will ease the implementation. The possibilities of new cache attacks and new attack vectors are also a part of future work. Different countermeasures to such statistical analysis could be researched. For instances, techniques like code obfuscation can be implemented but the level to which it should be done remains questionable.

6.2 Contributions

Attacks presented in this research are cache attacks on android smartphones namely, Nexus 5 and Samsung Galaxy S4. The contributions of this work are as follows:

- We demonstrate the applicability of cache attacks (PRIME+PROBE, EVICT+RELOAD) on a victim application making use of native code on Nexus 5. We reproduce the cache attacks on OpenSSL AES implementation by Gruss et.al [53] using *libflush*. We demonstrate EVICT+RELOAD and PRIME+PROBE cache attack on OpenSSL 1.0.1g and OpenSSL 0.9.7a implementations of AES, on Nexus 5 and Samsung Galaxy S4. We successfully reduce the secret key space of from 128 bits to 64 bits using Cache Template Attacks suggested by Spreitzer et al. [64].

- We are the first to demonstrate the usage of Cache Correlation Analysis and Key Rank Estimation techniques to extract key-related information from the cache-traces on Nexus 5 and Samsung Galaxy S4. Finally, we extract the secret key using $\approx$ 8000 cache traces on Nexus 5 and $\approx$ 700 cache traces on x-86 architecture.

- We are the first to demonstrate a cache attack implementation on a vulnerable AES implementation on an x-86 machine and then test its applicability on a custom white box implementation of AES.

Bibliography

[1] https://www.blackhat.com/docs/us-16/materials/us-16-Hornby-Side-Channel-Attacks-On-Everyday-Applications-wp.pdf.

[2] http://igoro.com/archive/gallery-of-processor-cache-effects.

[3] *Android software stack*, http://linux.softpedia.com/get/Programming/Interpreters/Android-SDK-32340.shtml.

[4] *Arm cache policies*, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/ch11s03.html.

[5] *Arm security part 1*, http://hiqes.com/android-security-part-1/.

[6] *Cache coherency*, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0434c/CJHBABIC.html.

[7] *Cache policies*, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/ch11s03.html.

[8] *Exploiting side channels in android*, http://cryptostackexchange.com/.

[9] *Exploiting unintended data leakage (side channel data leakage)*, http://resources.infosecinstitute.com/.

[10] *Inspector sca*, https://www.riscure.com/security-tools/inspector-sca/.

[11] *Memory management*, http://www.tldp.org/LDP/tlk/tlk.html.

[12] *Side channel attack against white-box cryptography on android*, https://github.com/edermi/papers/blob/master/Side-channel%20attacks%20against%20whitebox%20cryptography%20on%20Android/thesis.pdf.

[13] *These are not your grand daddy's cpu performance counters- cpu hardware performance counters for security*, https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardwar pdf.

[14] *Unintended data leakage,mobile top 10 2014-m4*, https://www.owasp.org/index.php/Mobile_Top_10_2014-M4.

[15] *Smartphone os market share, 2016 q3*, http://www.idc.com/promo/smartphone-market-share/os, 2016.

[16] Onur Aciiçmez, *Yet another microarchitectural attack:: Exploiting i-cache*, Proceedings of the 2007 ACM Workshop on Computer Security Architecture (New York, NY, USA), CSAW '07, ACM, 2007, pp. 11–18.

[17] Chris Wright Andrew Sloss, Dominic Symes, *Arm system developer's guide: Designing and optimizing system software.*

[18] AndrÃľ Seznec Anne Canteaut, CÃľdric Lauradoux, *Understanding cache attacks.*

[19] Ali Can Atici, Cemal Yilmaz, and Erkay Savas, *Remote cache-timing attack without learning phase.*, IACR Cryptology ePrint Archive **2016** (2016), 2.

[20] Daniel J Bernstein, *Cache-timing attacks on aes.*

[21] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo, *Aes power attack based on induced cache miss and countermeasure*, Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I - Volume 01 (Washington, DC, USA), ITCC '05, IEEE Computer Society, 2005, pp. 586–591.

[22] Andrey Bogdanov, Thomas Eisenbarth, Christof Paar, and Malte Wienecke, *Differential cache-collision timing attacks on aes with applications to embedded cpus.*, Springer.

[23] Joseph Bonneau and Ilya Mironov, *Cache-collision timing attacks against aes*, pp. 201–215, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[24] Joppe W Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen, *Differential computation analysis: Hiding your white-box designs is not enough*, International Conference on Cryptographic Hardware and Embedded Systems, Springer, 2016, pp. 215–236.

[25] Billy Bob Brumley, *Cache storage attacks*, pp. 22–34, Springer International Publishing, Cham, 2015.

[26] Billy Bob Brumley and Risto M. Hakala, *Cache-timing template attacks*, ASIACRYPT, 2009.

[27] Ketan Chavda, *Role of mobile apps in revolutionizing the world of iot*, (2016).

[28] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz, *Real time detection of cache-based side-channel attacks using hardware performance counters*, Applied Soft Computing **49** (2016), 1162–1174.

[29] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C Van Oorschot, *White-box cryptography and an aes implementation*, Springer.

[30] Joan Daemen and Vincent Rijmen, *The design of rijndael:aes the advanced*, Journal of Cryptology **4** (1991), no. 1, 3–72.

[31] Yoni De Mulder, Peter Roelse, and Bart Preneel, *Cryptanalysis of the xiao–lai white-box aes implementation*, International Conference on Selected Areas in Cryptography, Springer, 2012, pp. 34–49.

[32] Eli Dow, *Take charge of processor affinity*, https://www.ibm.com/developerworks/linux/library/l-affinity/l-affinity-pdf.pdf, 29/9/2005.

[33] Job de Haas Eloi Sanfelix, Cristofaro Mune, *Unboxing the white-box(practical attacks against obfuscated ciphers)*.

[34] Adrienne Porter Felt, Kate Greenwood, and David Wagner, *The effectiveness of application permissions*, Proceedings of the 2Nd USENIX Conference on Web Application Development (Berkeley, CA, USA), WebApps'11, USENIX Association, 2011, pp. 7–7.

[35] Catherine H. Gebotys and Brian A. White, *A sliding window phase-only correlation method for side-channel alignment in a smartphone*, ACM Trans. Embed. Comput. Syst. **14** (2015), no. 4, 80:1–80:22.

[36] Ben Gras and Kaveh Razavi, *Aslr on the line: Practical cache attacks on the mmu*, (2017).

[37] Marc Green, Leandro Rodrigues Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth, *Autolock: Why cache attacks on ARM are harder than you think*, CoRR **abs/1703.09763** (2017).

[38] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard, *Prefetch side-channel attacks: Bypassing smap and kernel aslr*, Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2016, pp. 368–379.

[39] Daniel Gruss, Clémentine Maurice, and Stefan Mangard, *Rowhammer. js: A remote software-induced fault attack in javascript*, Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2016, pp. 300–321.

[40] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard, *Cache template attacks: Automating attacks on inclusive last-level caches*, Proceedings of the 24th USENIX Conference on Security Symposium (Berkeley, CA, USA), SEC'15, USENIX Association, 2015, pp. 897–912.

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

[42] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone, *Guide to elliptic curve cryptography*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[43] Gernot Heiser, *Arm cache organization*, https://microkerneldude.wordpress.com/.

[44] Ralf Hund, Carsten Willems, and Thorsten Holz, *Practical timing side channel attacks against kernel space aslr*, Security and Privacy (SP), 2013 IEEE Symposium on, IEEE, 2013, pp. 191–205.

[45] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar, *Cross processor cache attacks*, Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, ACM, 2016, pp. 353–364.

[46] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall, *Side channel cryptanalysis of product ciphers*, Computer SecurityâĂŤESORICS 98 (1998), 97–110.

[47] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz, *Stealthmem: System-level protection against cache-based side channel attacks in the cloud.*

[48] Paul Kocher, Joshua Jaffe, and Benjamin Jun, *Differential power analysis*, Advances in cryptologyâĂŤCRYPTOâĂŹ99, Springer, 1999, pp. 789–789.

[49] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi, *Introduction to differential power analysis*, Journal of Cryptographic Engineering **1** (2011), no. 1, 5–27.

[50] Robert KÃűnighofer, *A fast and cache timing resistant implementation of the aes.*

[51] Cédric Lauradoux, *Collision attacks on processors with cache and countermeasures.*

[52] Steffen Liebergeld and Matthias Lange, *Android security, pitfalls and lessons learned*, Information Sciences and Systems 2013, Springer, 2013, pp. 409–417.

[53] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard, *Armageddon: Cache attacks on mobile devices*, 25th USENIX Security Symposium (USENIX Security 16) (Austin, TX), USENIX Association, 2016, pp. 549–564.

[54] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee, *Last-level cache side-channel attacks are practical*, IEEE Symposium on Security and Privacy (S&P), 2015.

[55] Dave Marshall, *Shared memory*, https://users.cs.cf.ac.uk/Dave.Marshall/C/node27.html.

[56] Michael Neve and Jean-Pierre Seifert, *Advances on access-driven cache attacks on aes*, International Workshop on Selected Areas in Cryptography, Springer, 2006, pp. 147–162.

[57] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang, *A refined look at bernstein's aes side-channel analysis*, Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security (New York, NY, USA), ASIACCS '06, ACM, 2006, pp. 369–369.

[58] Dag Arne Osvik, Adi Shamir, and Eran Tromer, *Cache attacks and countermeasures: The case of aes*, Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology (Berlin, Heidelberg), CT-RSA'06, Springer-Verlag, 2006, pp. 1–20.

[59] Dan Page, *Theoretical use of cache memory as a cryptanalytic side-channel.*

[60] Colin Percival, *Cache missing for fun and profit*.

[61] Romain Poussier, FranÃğois-Xavier Standaert, and Vincent Grosso, *Simple key enumeration (and rank estimation) using histograms: An integrated approach*, CHES, Springer, 2016, pp. 61–81.

[62] R. L. Rivest, A. Shamir, and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Commun. ACM **21** (1978), no. 2, 120–126.

[63] Andrew Sloss, Dominic Symes, and Chris Wright, *Arm system developer's guide: designing and optimizing system software*, Morgan Kaufmann, 2004.

[64] Raphael Spreitzer and Thomas Plos, *Cache-access pattern attack on disaligned aes t-tables*, pp. 200–214, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[65] Thomas Popp Stefan Mangard, Elisabeth Oswald, *Power analysis attacks: Revealing the secrets of smart cards (advances in information security)*, 1 ed., 2007.

[66] Junko Takahashi, Toshinori Fukunaga, Kazumaro Aoki, and Hitoshi Fuji, *Highly accurate key extraction method for access-driven cache attacks using correlation coefficient*, Australasian Conference on Information Security and Privacy, Springer, 2013, pp. 286–301.

[67] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi, *Cryptanalysis of des implemented on computers with cache*, Springer.

[68] Rob van der Meulen, *Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016*, (2017).

[69] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida, *Drammer: Deterministic rowhammer attacks on mobile platforms*, Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2016, pp. 1675–1689.

[70] Nicolas Veyrat-Charvillon, Benoît Gérard, Mathieu Renauld, and François-Xavier Standaert, *An optimal key enumeration algorithm and its application to side-channel attacks*, International Conference on Selected Areas in Cryptography, Springer, 2012, pp. 390–406.

[71] Michael Weiß, Benedikt Heinz, and Frederic Stumpf, *A cache timing attack on aes in virtualization environments*, International Conference on Financial Cryptography and Data Security, Springer, 2012, pp. 314–328.

[72] Wikipedia, *Cache coherence — wikipedia, the free encyclopedia*, 2017.

[73] ———, *Cpu cache — wikipedia, the free encyclopedia*, 2017.

[74] Brecht Wyseur, *White-box cryptography: hiding keys in software*.

[75] Yaying Xiao and Xuejia Lai, *A secure implementation of white-box aes*, Computer Science and its Applications, 2009. CSA'09. 2nd International Conference on, IEEE, 2009, pp. 1–6.

[76] Zhao Xinjie, Wang Tao, Mi Dong, Zheng Yuanyuan, and Lun Zhaoyang, *Robust first two rounds access driven cache timing attack on aes*, Computer Science and Software Engineering, 2008 International Conference on, vol. 3, IEEE, 2008, pp. 785–788.

[77] Yuval Yarom and Katrina E Falkner, *Flush+ reload: a high resolution, low noise, l3 cache side-channel attack.*, USENIX Security Symposium, vol. 2014, 2014, pp. 719–732.

[78] Yuval Yarom, Daniel Genkin, and Nadia Heninger, *Cachebleed: A timing attack on openssl constant time rsa*, Journal of Cryptographic Engineering **7** (2017), no. 2, 99–112.

[79] YongBin Zhou and DengGuo Feng, *Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing.*

Appendix

A.1 Processor Cache Effects

It is important to understand that how interaction [2] between processor and cache memory influences the performance of a program or of an algorithm.

A.1.1 Impact of Cache Lines

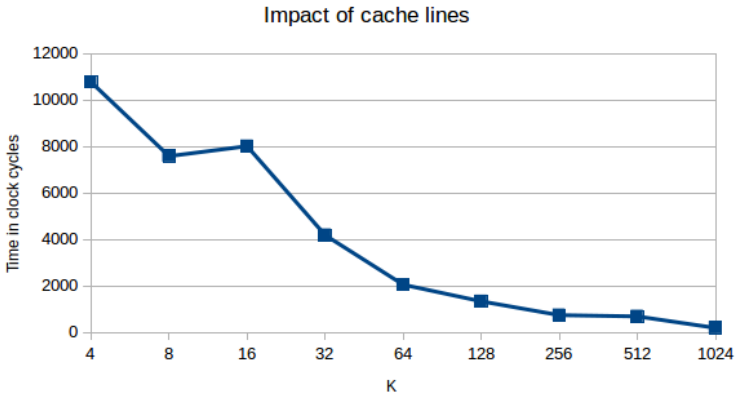


Figure A.1: Step-size(k) vs Access times(clock cycles)

Understanding of cache lines can be important for certain types of program optimizations. For example, the alignment of data may determine whether an operation touches one or two cache lines. As can be seen in the Figure: A.1, till the time step-size(k) was within one cache-line size i.e, 16 (as 16 ints take up 64 bytes), there was no significant performance change. The reason behind such behavior is that today's CPUs do not access memory byte by byte. Instead, they fetch memory in chunks of (typically) 64 bytes, called cache lines. When a particular memory location is read, the entire cache line is fetched from the main memory into the cache. And, accessing other values from the same cache line is cheap. Since 16 ints take up 64 bytes (one cache line), for-loops with a step-size between 1 and 16 will touch all of the cache lines in the array. But once the step-size is 32, we'll touch roughly every other cache line, and once it is 64, only every fourth.

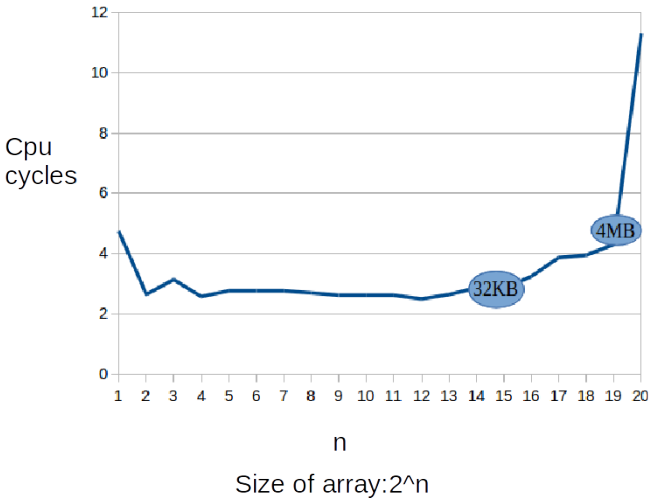


Figure A.2: Access Times vs Array Size

A.1.2 Impact of L1-L2 Sizes

Figure: A.2 reflects at what size of an array the overall access time increases with a sudden spike. The horizontal axis shows the size of the array (2^n) where n is $1, 2, ..., 20$ and the vertical axis shows the cpu cycles consumed to access array elements. Following can be inferred from Figure: A.2:

- L1 cache is somewhere around 32 KB
- L2 cache is 256 KB
- L3 cache is 4MB

The jumps in the access times of the array help us understand how access time of each element increases with the increase in the size of the array. The reason behind is that, beyond the size of the cache, the cache cannot keep the entries of an array and hence it will have to fetch them from memory when requested. The above-mentioned behavior is the behavior of an x86 architecture based machine. For our device Nexus 5, the results are pretty much the same with spikes on 16 KB and 2 MB positions as L1 is 16 KB and L2 is 2MB on Krait 400.

A.1.3 Impact of Cache Associativity

Direct mapped caches can suffer from conflicts, for e.g. when multiple values compete for the same slot in the cache, they keep evicting each other out, and the hit rate plummets. On the other hand, fully associative caches are complicated and costly to implement in the hardware. N-way set associative caches are the typical solution for processor caches,

as they make a good trade off between implementation simplicity and good hit rate.

For example, the 2MB L2 cache on Nexus 5 is 8-way associative. Each of the 4096 sets will have 8 ways/lines in the cache. So, the lowest 12 bits of the memory block index will determine which set the block belongs to (2^{12} = 4,096). As a result, cache lines at addresses that differ by a multiple of 262,144 bytes (4096 * 64) will compete for the same slot in the cache. The cache on Nexus5 can hold at most 8 such cache lines. For the effects of cache associativity to become apparent, we need to repeatedly access more than 8 elements from the same set as this will result in the eviction of the desired set. On the basis of this property, we will understand how cache eviction is performed without any dedicated instruction and will try to achieve the maximum possible eviction rate.

A.1.4 Memory Organization

In Figure: A.3 a set associative cache memory and the memory organization are shown. As shown in the Figure: A.3, the cache memory is divided into ways/lines (W) and sets(S) and its size is decided by the processor and the cache design. Each memory block is mapped to a specific cache set , for e.g, when an S-box element is allocated in the memory (Figure: A.4), each element of S-box table is cached in corresponding cache set when a cache miss occurs [66].

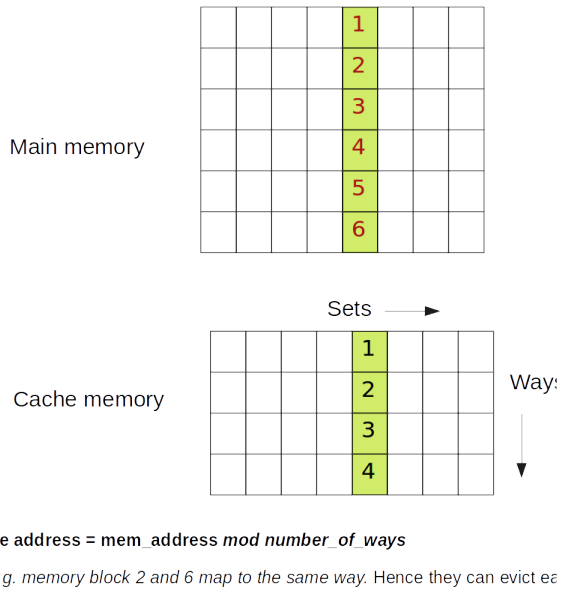


Figure A.3: Mapping from main memory to cache

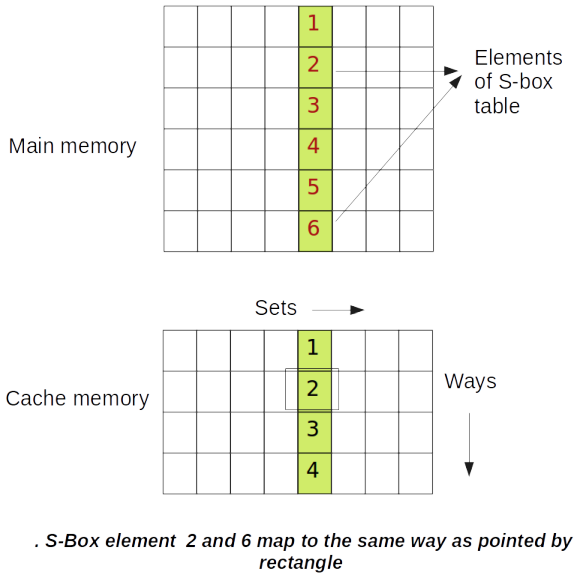


Figure A.4: S-box elements mapped from main memory to cache

A.1.5 Impact of CPU Affinity

If we are running time-sensitive or deterministic processes, there is a reason to be interested in CPU affinity [32]. For example, hard affinity can be used to specify one processor on an eight-way machine, while allowing the other seven processors to handle all the normal scheduling needs of the system. This action ensures that our long-running, time-sensitive application gets to run, and also allows other application(s) to monopolize the remaining computing resources.

A.2 Types of Cache Misses and their utility

Cache eviction is performed to create some space for new entries in the cache when the cache is full. It can also be triggered manually by exploiting a specific category of cache misses which is called **Conflict Misses**. Cache misses [18] can occur because of following three reasons:

- Whenever a data is referenced for the first time, it has to be fetched from the memory and hence there is a cache miss. This is termed as *Compulsory Misses*
- For a given size of a cache, any data which is more than the limited size of the cache would result in a cache miss. This is termed as *Capacity Misses*

- If one or more memory access references a particular address then there arises a conflict and data present in the cache earlier would be evicted. This is called *Conflict Misses*

The 3^{rd} category of cache misses can be effectively used to perform cache eviction. The data from the cache can be evicted using several approaches which are only applicable to Least Recently Used (LRU) replacement policies [41] and thus, are not suited for ARM CPUs. However, Spreitzer and Plos [64] proposed an eviction strategy for ARMv7-A devices that tries to overcome the pseudo-random replacement policy by accessing more addresses than there are ways per cache set. Gruss et al.[53][40] demonstrated an automated way to find fast eviction strategies on ARM.

A.2.1 L1 and L2 cache misses

L1 cache is local to the processor's cores and hence in order to perform cache attacks both spy and victim applications have to run on the same core. The size of L1 cache also plays a major role in terms of eviction and cache misses. If the size of L1 is small then it will result in more cold start misses (2). On the other hand, in the case of L2 cache, since it is used for both data and code, there will be some inevitable cache collisions (and line evictions) caused by the instruction fetching activity. However, since L2 cache is bigger in size, the possibility of false positives during eviction gets reduced. Attacks exploiting the L2 or LLC cache in android smartphones have demonstrated the recovery of sensitive information like cryptographic keys [45][77].

Another problem arises from the processor's capability for hardware prefetching: If a series of cache misses occur, in an arithmetic progression, within a single page, then the cache will recognize this as a data stream and prefetch two additional cache lines. The solution is to access the cache lines in an irregular manner (i.e., to follow a de Bruijn cycle instead of accessing the lines in an arithmetic progression). Thus, hardware prefetcher will not get activated as well. In most of the past attacks [60][25][22], the initial state of cache (empty, loaded or initialised) is not clearly stated and hence, the different type of cache misses (cold, conflict and capacity) arises [60]. However, it should be mentioned that most of the cache attacks make use of cold or conflict misses. This is due to the fact that symmetric encryption techniques are designed to have a relatively high speed. Hence, all the operations must be implemented by lookup tables whose total size do not exceed the cache capacity.

A.3 Eviction Results: Nexus 5

Table:A.1 summarizes different eviction strategies for Krait 400 (Nexus 5). Column 1 defines the number of addresses being accessed using Rowhammer Eviction Strategy [39]. Column 3 signifies the eviction rate. As can be seen from the table that row 1 and row 3, despite almost the same number of different addresses being accesses in a loop, there exisits a significant difference in the eviction rate. The reason behind is the number of accesses performed to those different addresses which facilitate the eviction and successfully tricks the PLRU policy.

Table A.1: Eviction Results: Nexus 5

Number of Addresses (N)	Number of Accesses (A)	Eviction Rate (in %)
13	1	79.97
12	3	100
10	2	100

A.4 White-Box Use cases

Example 1: Digital Signature (Figure: A.6) and Example 2: Content Protection (Figure: A.5) describe the utility of WBC in their functionality. WBC is an integral part of these services offering Content Rights Management on mobile devices such as smartphones.

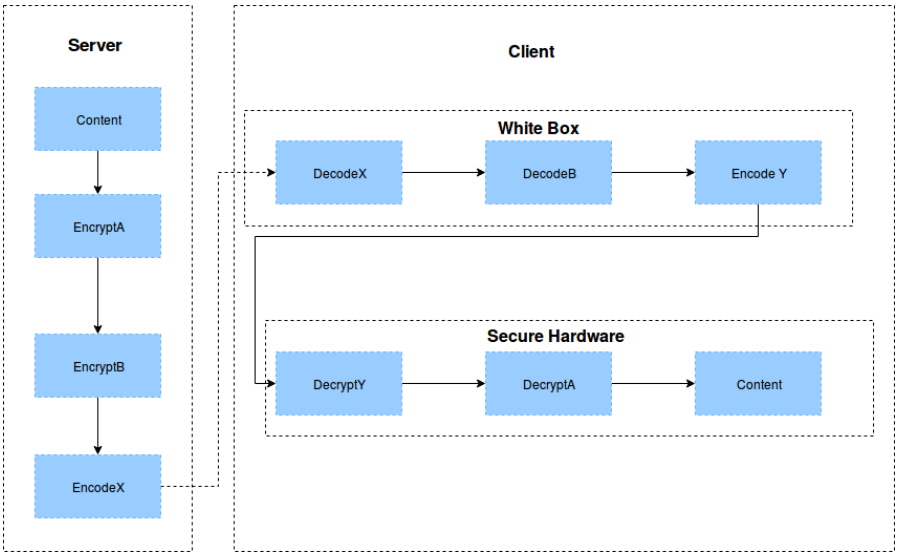


Figure A.5: Content Protection using White-Box

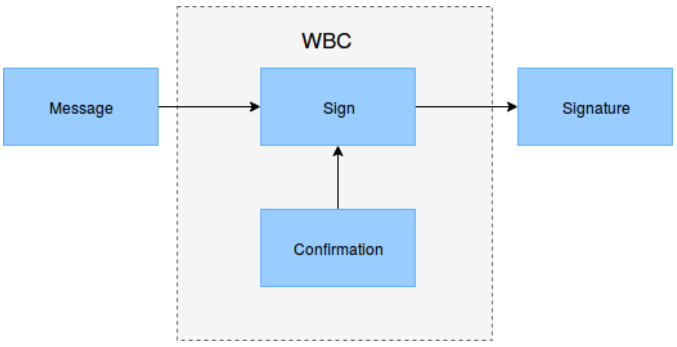


Figure A.6: Digital Signature using White-Box

A.5 First Order Analysis and Key Rank Estimation

Figure:A.7 shows first-order analysis (Correlation Analysis) of cache traces from an attack on AES last round. All three steps namely: Trace acqusition, Binary Threshold Filtering, and Correlation Analysis are shown. Figure: A.8 demonstrates the usage of key rank estimation on the correlation analysis. Figure: A.9 demonstrates how key entropy reduces with every 100 cache traces. Figure:A.10 shows how correlation analysis helps in full recovery of key. Additionally, figure: A.11 explains how subsets of 8000 cache traces each, respond to the key extimation technique. Finally, in order to find the bounds on the number of cache traces required, we show the effect of Minimum-Maximum-Average module in Figure: A.12.

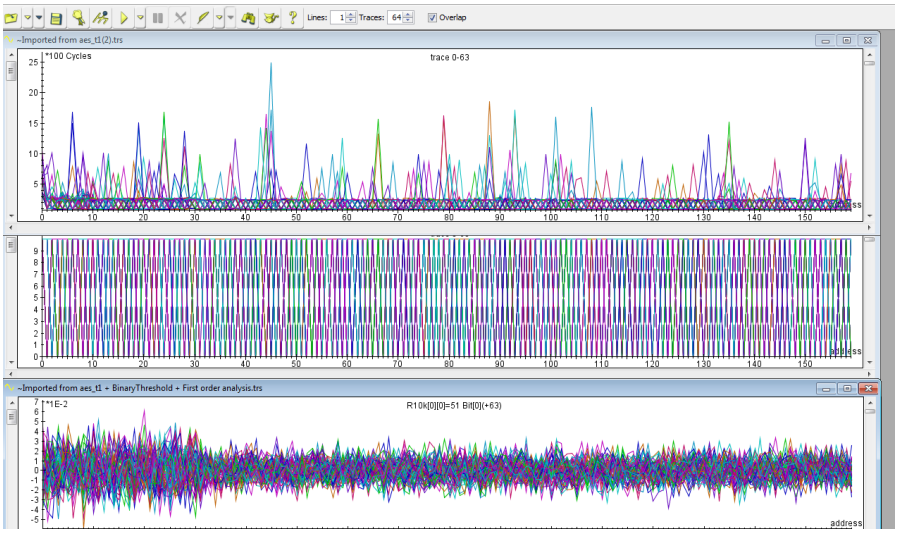


Figure A.7: Steps of Correlation Analysis of cache attack on last round of AES

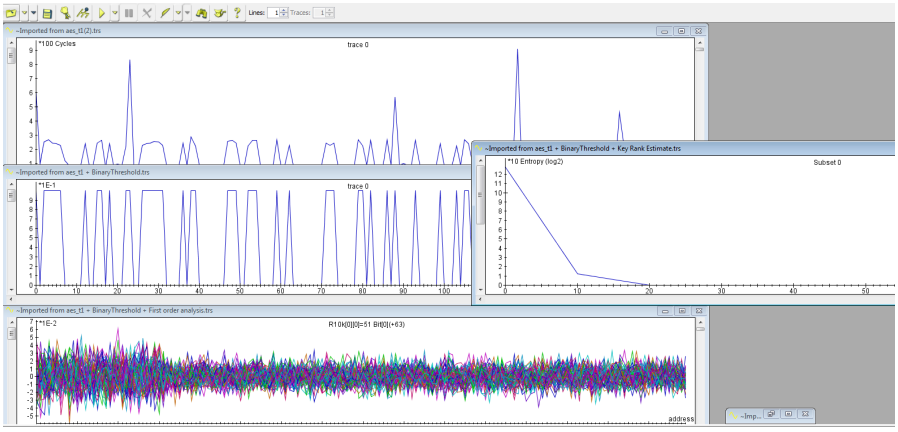


Figure A.8: Key rank estimation of cache attack results on AES last round

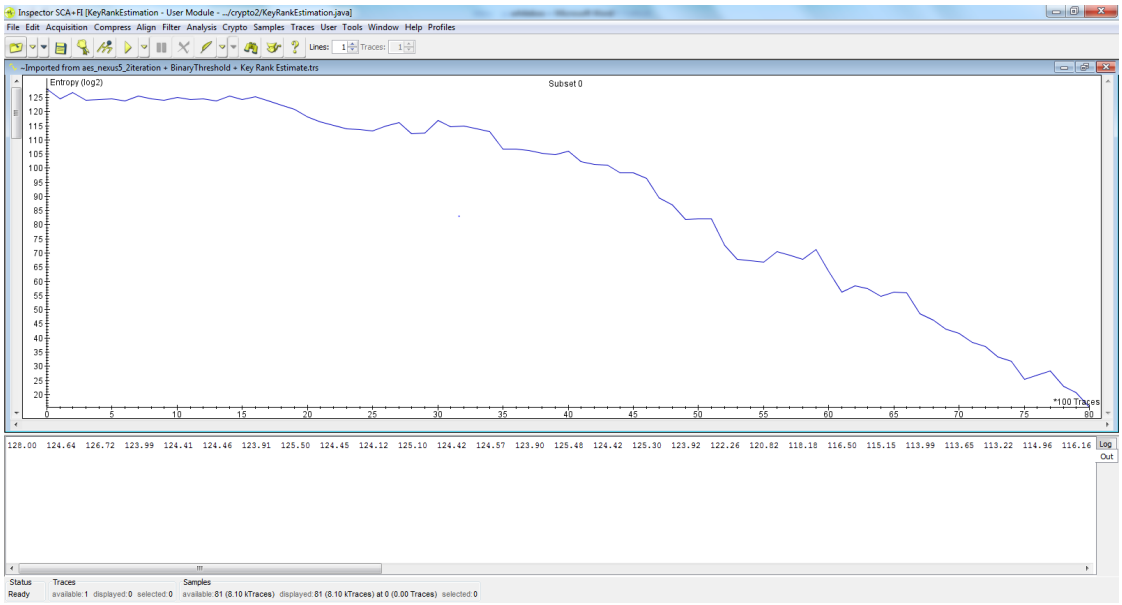


Figure A.9: Nexus 5: Key Rank Estimation on 8000 traces

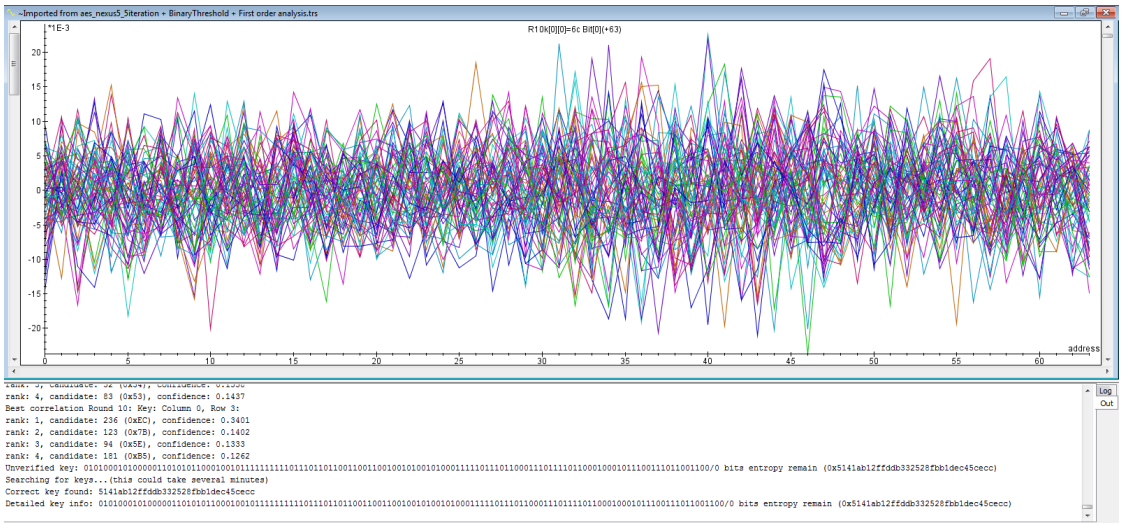


Figure A.10: Nexus 5: Correlation Analysis 320000 cache traces

build

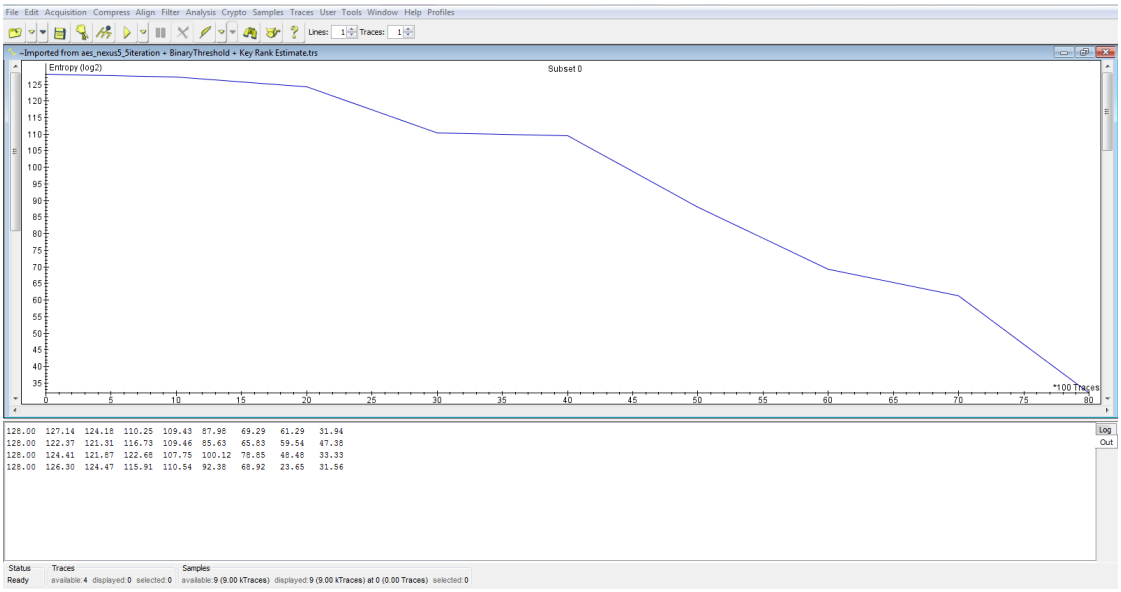


Figure A.11: Nexus 5: Key rank estimation of subsets (8000 cache traces each)

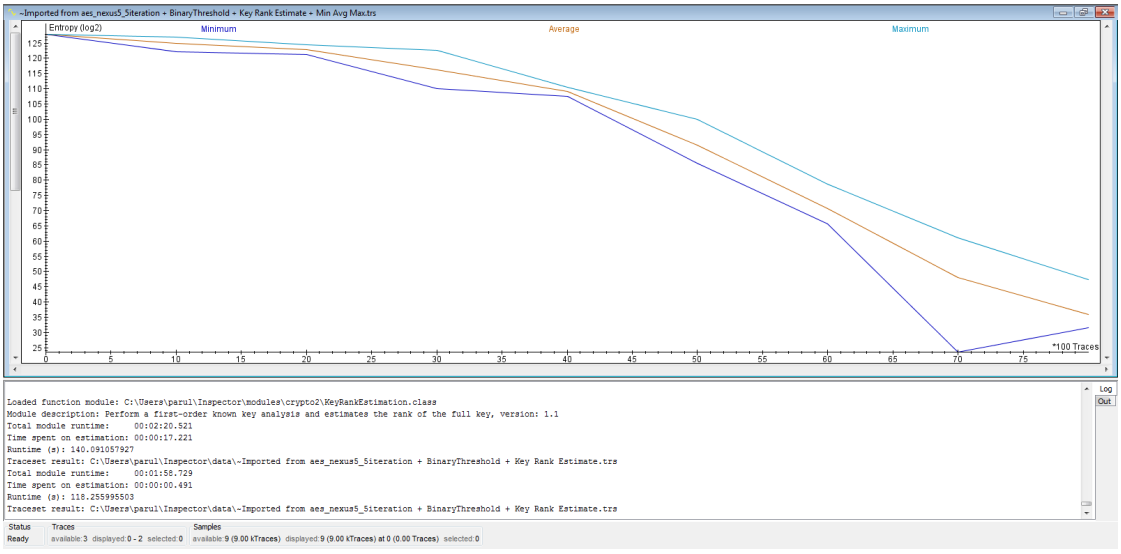


Figure A.12: Nexus 5: Minimum-Average-Maximum of subsets (8000 cache traces each)

build

Cache Correlation Analysis and Key Rank Estimation on Android Smartphones

Parul Gupta, Christian Doerr, and Ruben Muijrers

Cyber Security Group, Delft University of Technology
Delft, The Netherlands
{p.gupta-3}@student.tudelft.nl
{c.doerr}@tudelft.nl
muijrers}@riscure.com

Abstract. Android smartphones collect and compile a huge amount of sensitive information which is secured using cryptography. There is an unintended leakage of information during the physical implementation of a cryptosystem on a device. Such a leakage is often termed as side channel and is used to break the implementation of cryptographic algorithms. In this work, we utilize cache memory based side channels on android smartphones to retrieve crypto-process information. These side channels are based on the information leakage through the operating system, micro-architecture of the processor and the state of the processor's memory cache. We demonstrate the retrieval of data dependent memory access patterns using a spy application running in the background to recover the full secret key of cryptographic primitives such as AES T-table implementation in OpenSSL, all that would be necessary is a rogue app downloaded from an app store that is run under normal privileges. We show that a mathematical correlation which depends on the guessed key, can be utilized to recover the *complete* key in access-driven cache attacks (CAs). We show the effectiveness of the proposed method using access time measured in noisy environments. We analyze the changes in the correlation values with the number of plaintexts/ciphertexts for a successful attack using key estimation. Furthermore, we discuss and demonstrate the applicability of cache memory based side channel attacks on a white-box implementation of AES.

Keywords: Cache Attacks, Android, OpenSSL AES, Key estimation, Correlation Analysis

1 Introduction

With $\approx 87\%$ market share [4], Android clearly dominates the smartphone market. One of the reasons behind such a huge success is that android is considered to be "open and free". Also for developers, it allows them to incorporate already available third party code in their applications which can be confirmed by the heavy usage of native libraries among the most popular applications. However, including third party code in an application can have some severe

consequences both on the user and the developer of the app. In practice, cryptographic primitives and cryptographic protocols are implemented to protect user's information from malicious applications and 3^{rd} party native code. Smartphones use cryptographic keys to protect sensitive information such as health records and banking passwords. These techniques intrinsically make use of standardized cryptographic algorithms such as Advanced Encryption Standard (AES) etc. In cryptographic implementations, the secret key directly affects the emitted side-channel information for e.g. power usage patterns or memory access patterns. Hence, observations made on this leaked data can eventually lead to the revelation of the secret key. Such malicious leakage to untrusted third-party applications which can exploit these side-channels is one of the key challenges to the smartphone data security and privacy.

Typically, Android uses a kind of Unix Sandboxing method to run the applications. Usually, mobile applications run with different permissions and privileges and the low-level implementation of machine in collaboration with OS provides for desired access control. For example, each application that is installed on an Android device is assigned its own unique user identifier (UID) and group identifier (GID). This behavior is different than conventional Linux, where applications that are shared by a user are run under user's context. Specifically, it means that the processes running as separate users cannot interfere with each other such as sending signals or accessing one another's memory space. Android applications execute within a register-based VM, DalvikVM which relies on functionality provided by a number of supporting native code libraries. Similar to Java VM, the DalvikVM interfaces with low-level native code using Java Native Interface (JNI). Once a process or an application starts when an Android device boots is the Zygote process. It is responsible for loading libraries used by the Android Framework. It also acts a loader for each Dalvik process and prevents the repetitive loading of the Android framework and its dependencies when starting applications. As a result, core libraries, core classes and their corresponding heap structures are shared across instances of the DalvikVM. For e.g., a device running Android 4.3 contains more than 200 shared libraries. Much of the low-level functionality is implemented in these shared libraries which are developed in native code (are a part of well-known open source projects). The usage of native code makes them prone to memory corruption vulnerabilities, side-channel leakage etc.

A potential leakage in the form of execution footprints [29] leaks timing information such as, access time to perform a look-up in a table while performing a crypto operation. Such a side-channel originally stems from the microarchitectural structure of the underlying microprocessor. In 2006, Osvik et.al [29] showed how a low-level implementation detail of modern CPUs, namely the structure of memory caches, leads to cross-process information leakage between processes running on the same processor. In essence, the cache forms a shared resource which all processes compete for, and it thus affects and is affected by

every process. While the data stored in the cache is protected by virtual memory mechanisms, the metadata about the content of the cache, and hence the memory access patterns of processes using that cache, are not fully protected [23][30][37][24]

From the perspective of side-channel attacks, most of the attacks on a smartphone require an adversary to have physical access to it or at least have a cable or probe in close proximity to the device while it is performing some cryptographic operation. Usually the time to attack an application is dependent on the application specification for e.g. time to hack RSA [33] would be different than that required for ECC [34] occurs when a developer accidentally places sensitive information or data in a location on the mobile device that is easily accessible by other apps on the device. For instance, a developer's code processes sensitive information supplied by the user or the backend. During that processing, a side-effect (that is unknown to the developer) results in that information being placed into an insecure location such as cache etc. on the mobile device that other apps on the device may have open access to. Under these assumptions, we can safely assume that one malicious app can extract valuable information about other apps running on the same device.

The paper is organized as follows. In Section 2 we provide the technical background related to cache attacks and challenges faced on ARM. After that we introduce our threat model in Section 3. In Section 4 we give our formal outline of a cache attack. We cover the cache attacks on the first round [12] and on the last round of AES. In Section 5, we discuss about leakage models and cache correlation analysis. We show the application of key estimation techniques on the generated cache traces in Section 6. In Section 7, we discuss the impact of various attack parameters on the intensity of the attack. Then, we present and discuss our results and provide with subsequent application of our technique on white-box cryptosystems. We finish with remarks about future research.

2 Related Work

In 1999, Simple Power Analysis (SPA) and Differential Power Analysis (DPA) was introduced by Kocher et al. [24]. According to them, an attacker can extract cryptographic keys by studying the power consumption of a device. They [23] further explained that there is a leakage of information from computers and chipsets about the operation they execute. They utilized the power consumption measurements to find secret keys from tamper resistant devices.

A different side channel on modern computing architectures is introduced by the memory hierarchy that stores subsets of the computer's memory in smaller but faster memory units, so-called caches. Cache side-channel attacks exploit the different access times of memory addresses that are either held in the cache or the main memory. Kelsey and Kocher et al. [21][23] were the first to discuss the theo-

retical cache attacks. Page and Tsunnoo et al. [30][37] discussed the applicability of cache attacks on DES. Based on his work, Bernstein [6] demonstrated complete AES key recovery from known-plaintext timings of a network server. Hund et al. [20] demonstrated how an adversary can implement a generic side channel attack against the memory management system to deduce information about the privileged address space layout. Gullasch et al. [19] attacked the L1 cache and demonstrated the exploitation of shared memory to mount cache attacks. The Evict+Time and Prime+Probe techniques by Osvik et al. [29] explicitly targeted cryptographic algorithms. While Herath et al. [3] discussed the CPU Hardware Performance Counters for Security, Chiappetta et al. [11] demonstrated the real time detection of cache-based side channel attacks using hardware performance counters. Yarom and Falkner introduced Flush+Reload attack in 2014 where the target was L3 cache instead of L1 cache. It allows an attacker to determine which specific parts of a shared library or a binary executable have been accessed by the victim with an unprecedented high accuracy. Based on this work Gruss et al. [26] demonstrated the possibility to exploit cache-based side channels via cache template attacks in an automated way and showed that besides efficiently attacking cryptographic implementations, it can be used to infer keystroke information and even log specific keys. Zhao et al. [42] presented an access-driven attack on the first and second round of the AES encryption. Laradoux et al. [25] explained about the collision attacks on processors with cache and countermeasures.

However, it is also possible to induce hardware faults by software, and thus from a remote location, if the device could be brought outside of the specified working conditions. In 2014, Kim et al. [22] demonstrated that accessing specific memory locations at a high repetition rate can cause random bit flips in Dynamic Random-Access Memory (DRAM) chips. Since DRAM technology scales down to smaller dimensions, it is much more difficult to prevent single cells from electrically interacting with each other. They observe that activating the same row in the memory corrupts data in nearby rows. Gruss et al. [26][17] showed that such bit flips can also be triggered by JavaScript code loaded on a website. However, this attack can only be demonstrated on Intel and AMD systems using DDR3 and modern DDR4 modules. Brumley et al. [10][9] carried out the attack on live cache-timing data and cache storage data. In 2015, Seaborn demonstrated that cache side channel attack could be exploited for privilege escalation. Weiss et al. [39] and Bogdanov et al. [7] attacked ARM7 microcontrollers and ARM Cortex-A8 processors. Van der Veen et al. [15][38] investigated the rowhammer bug [22] on ARM-based devices as well. They successfully triggered the bug on multiple mobile devices and built a root exploit for Android. Key rank estimation techniques came as a major breakthrough in the field of estimation of security of a cryptographic implementation. Vincent et al. [14] [32] explained a simple key enumeration and rank estimation technique using Histograms.

3 Technical Preliminaries

In this section, we provide a brief description about processor cache memory and cache attacks.

3.1 Cache Memory

A cache memory [41][40] is a small, volatile and fast array of memory placed between the processor and main memory. It is equipped with additional features to cater to high throughput requirements of a processor. Thus, when a processor requests data that already present in the cache memory, it is brought to the registers within a single clock cycle without stalling the pipeline and results in a cache hit. If not present in the cache, this results in the cache miss, and the desired data is fetched from Non-Volatile Memory (NVM), and the entire line containing the desired data is loaded into the cache. Cache memory is a smaller and faster storage area in comparison to the main memory and therefore many different addresses in the main memory are mapped to the same cache entry (associative caches). Associative caches in modern processors consist of cache sets (S) which contains cache lines (B) with (W) bytes each. Hence the size of cache is (S.B.W). The cache architectures are optimized to minimize the number of cache misses for typical access patterns but can be easily manipulated. Depending on the cache replacement policy, such property can be used to perform manual cache eviction and monitor the cache behavior. Additionally, a data item residing in the cache (cache hit) is retrieved much faster than a data item that is not in the cache (cache miss) and the difference in access times is measurable. Thus, cache hit/miss ratio can also be exploited to retrieve the memory access patterns during the execution of a process (in our case a crypto process).

3.2 Cache Attacks on ARM

In this section we discuss cache attacks [13][19][35][16][17][5][29] and their applicability on ARM. A cache attack exploits the cache behavior of a cryptosystem by obtaining the execution time and/or power consumption variations generated via cache hits and misses. Cache analysis techniques enable an unprivileged process to attack another process, e.g., a cipher process, running in parallel on the same processor as done in past research work (Section: 2). The memory accesses of software cryptosystems, especially S-box based ciphers like DES and AES, employ key-dependent table lookups, indices of which are simple functions of the key and the plaintext. Revealing these memory access patterns, i.e. lookup indices via cache statistics and the knowledge of the processed message makes it relatively easy to break these ciphers. Such attacks are known as access driven cache timing attacks [29] [30]. They utilize the particularities of microcontrollers and microprocessors with the cache memory which frequently exhibit data-dependent timing. As discussed before, it is a result of one of the processor optimizations where variable speed-up is required. Different execution times for different inputs amounts to cache hits and misses in specific cache sets depending

upon the implementation of the crypto algorithm involving S-box invocations in software. When the inputs to S-boxes are key-dependent, this timing information frequently turns out to be sufficient to recover the entire key. If the access is performed to cached data (S-box entries), then it requires less time than the data which is not present in the cache. The penultimate purpose of these attacks is to determine which cache lines or cache sets have been accessed during the encryption. Hence, knowledge of the location of the pre-computed S-boxes or T-tables within the memory as well as information about the cache architecture is necessary. However, fewer measurement samples are required in comparison to the time-driven attacks in order to recover the secret key [29].

Historically, these attacks [30][29] can be further divided into following 3 categories:

- **EVICT + TIME:** The attacker measures the time it takes to execute a piece of victim code. Then attacker flushes part of the cache, executes and times the victim code again. The difference in timing tells whether the victim uses that part of the cache.

- **PRIME + PROBE:** The attacker accesses memory to fill part of the cache with his own memory and waits for the victim code to execute. This is called the Prime Step. Then the attacker measures the time it takes to access the memory that he carefully placed in the cache before. This is called the Probe Step. If the access time is higher than a certain threshold for certain cache line, then we know that the victim process evicted those cache lines from the cache. If the access time is less than a certain threshold, then it becomes clear that victim did not access those lines or evict those cache lines.

- **FLUSH + RELOAD:** The flush and reload attack utilizes the fact that processes often share memory. By flushing a shared address, then wait for the victim and finally measuring the time it takes to access the address an attacker can tell if the victim placed the address in question in the cache by accessing it.

The rising popularity and usage of smartphones in our everyday life clearly states the need for the investigation of such cache attacks on modern smartphones. It also becomes important to study the assisting techniques which make these attacks more viable. To this aim, we discuss the applicability of FLUSH+RELOAD and PRIME+PROBE attacks on ARM-v7 based smartphones. However, it is important to mention that ARM-v7 does not provide for unprivileged flush instructions unlike x86. Therefore, in place of cache flush, we perform manual cache eviction as suggested in [18] [26]. Hence, instead of FLUSH+RELOAD, the connotation becomes EVICT+RELOAD. We can perform these attacks on an unrooted Android smartphone or without any privileged access as well with the help of libflush [27]. In contrast to x86, there are several challenges on ARM-v7 CPUs which can be enumerated as follows :

1. Non-inclusive/exclusive cache
2. No unprivileged cache flush instruction
3. Pseudo random cache replacement policy
4. No unprivileged access to cycle accurate counter

Using libflush [27], not only could we overcome all the above mentioned challenges on ARM but could also attack white-box cryptosystems. In the next section, we will describe our threat model which implements EVICT+RELOAD and PRIME+PROBE attacks on OpenSSL 1.0.1g AES implementation on ARM (Section: 5.1).

4 Threat Model

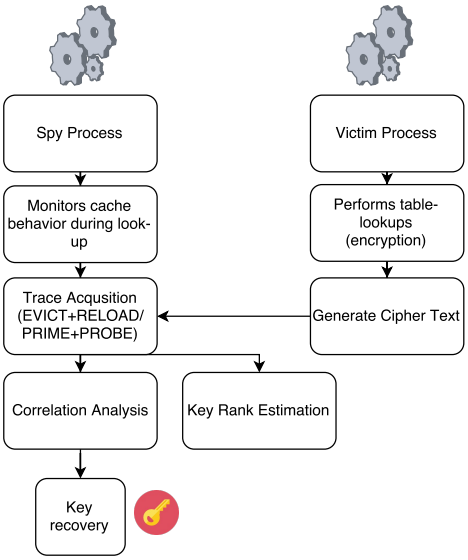


Fig. 1: Threat Model

Our access-driven cache attack on software implemenation of AES is inspired by the works of Osvik et al. [29], Neve et al. [28], Spreitzer et al. [35] and Bonneau et al. [8]. The threat model 1 comprises of two processes running on the same or different processors. The CPU affinity is important from the aspect that there should be no context switching while are processes are running on their respective CPUs. In this attack neither do we have any information regarding when a T-table is accessed during an AES round nor do we get any information about the order of accesses within one measurement. We cannot identify any distinction between AES rounds. For EVICT+RELOAD attack, the prerequisite is to map a shared library (*libcrypto.so*) used by the victim crypto process into attacker's address space. We assume the knowledge of position of T-tables in the memory,

some preprocessing steps are required as mentioned in [35] and [29]. The access times are calculated using any of the following timing mechanism: cycle counter (PMCCNTR), *perf* or *monotnonic clock*. After the collection of access-times corresponding to the cache sets, we can map them to cache hit or miss on the basis of a pre-calculated threshold. In the previous cache attack implementations [35] [26] [10], it was a mandatory step, however, in our implementation this step can be omitted as it does not effect the ultimate aim of full key recovery. It does affect the number of traces required for the key recovery as the useful information about access times gets affected by noise a lot. There is no such perquisite for shared memory in PRIME+PROBE and the remaining steps are explained in Section: 4.3. We implemented cache attacks on our x-86 and ARM based test devices (Section: 5.1) using the generic attack strategy mentioned in Figure: 2. The threat model used to perform the following two attacks on OpenSSL T-table AES implementation (1.0.1g) is shown in Figure: 1.

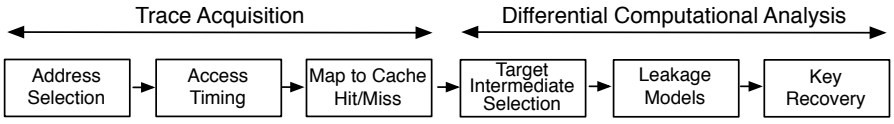


Fig. 2: Attack steps

4.1 Attack Assumptions

In order to simplify the descriptions and analysis of our access-driven cache attacks [18][29][10][29] on ARM, we will start with the following assumptions based on the work of Zhao et al. [42] :

- The attacker uses uniformly distributed plaintexts.
- The attacker has access to an accurate and high performance timing mechanism for example cycle counter on ARM, *perf* or *monotonic clock*.
- The attacker operates synchronously with the victim process.
- The attacker knows the cipher text.

4.2 EVICT + RELOAD Attack

EVICT+RELOAD (Figure: 3) is a variant of FLUSH+RELOAD as instead of flushing the cache line, we perform eviction. In the first step, spy process maps a shared library or binary into its own address space. The second step is to evict a cache entry from the shared memory. Then the victim is scheduled again on the processor. In the final step, the spy process checks whether the evicted line is loaded or not again.

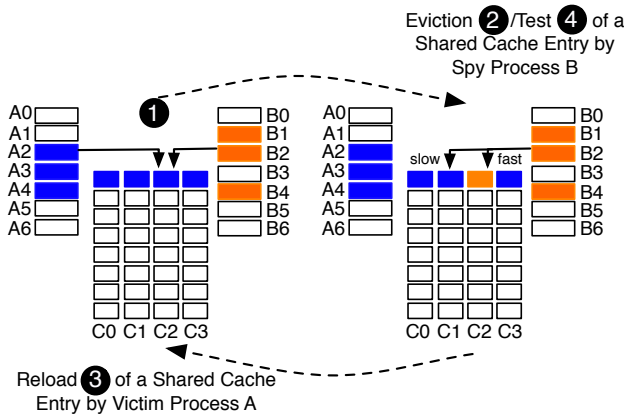


Fig. 3: EVICT+RELOAD cache attack

4.3 PRIME + PROBE Attack

As mentioned before, PRIME+PROBE (Figure: 4) does not involve the usage of shared libraries. The first attack step is that a spy process primes the cache memory with its own data. In the following step, the victim is scheduled and it evicts certain cache lines during execution. The attacker process checks data to determine if the primed sets were accessed or not.

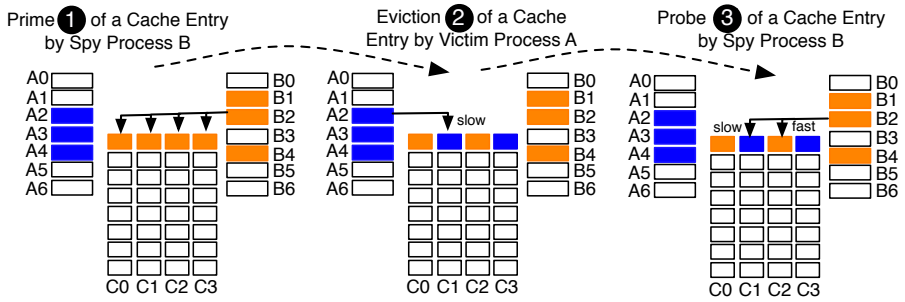


Fig. 4: PRIME+PROBE attack

5 Experiments

5.1 Target Devices

We aim to perform cache based side channel attacks on Nexus 5 (quad-core KRAIT 400) and Samssung Galaxy S4 (quad-core KRAIT 300) with the following (Table: 1) cache configurations.

Level of cache	Cache Size	Associativity	Cache Line Size	Inclusiveness	Device
L1	**2x16KB (per core)**	4-way	64 Bytes	**Non-inclusive, exclusive**	Nexus 5
L2	**0.5MB (per core)**	8-way	64 Bytes	**Shared, unified**	
L1	**2x16KB (per core)**	4-way	64 Bytes	**Non-inclusive**	Samsung Galaxy S4
L2	**0.5MB (per core)**	8-way	128 Bytes	**Shared, unified**	

Table 1: Cache organization

5.2 Preliminary Analysis and Notations

Cache Attack (EVICT+RELOAD) on AES 1^{st} round: The first round attack is performed on sbox out of 1^{st} round of AES (on both versions 0.9.7a and 1.0.1g) using strategy explained in [35]. In the first round of AES, the following operations takes place for i=1,..,16:

$$x_i = p_i \oplus k_i \tag{1}$$

Here, x_i is the state byte which is used to perform T-table lookup for the next-round and p_i is a plaintext byte and k_i is a key byte. Hence, if we know the plaintext, then we can guess the upper 4 bits of the address being accessed, as explained in [29][8][31][35].

Cache Attacks on AES last-round: Our implemantation works on T-table AES implementation (OpenSSL 1.0.1g). In OpenSSL 1.0.1g, all four tables namely; T_0,T_1,T_2,T_3 are used in the 10^{th} round. We perform $n > 8000$ number of encryption for each cache set occupied by the T-tables. For demonstration, we use OpenSSL 1.0.1g (32 bit version) on Android , however the attack should be scalable to other versions as well. The total space occupied by $T_0 - T_3$, tables is 4KB. Hence, the number of cache sets occupied by T-tables is 64, as the size of each cache line is 64 bytes and each set contains 8 cache lines (8-way associativity). For EVICT+RELOAD, we evict a specific cache set and perform encryption. If the evicted cache set is used by the encryption, then it will be fetched again from the memory, otherwise not. In the last and final step we access the evicted address again to check if it is loaded during the encryption step. We store the ciphertext and corresponding CPU cycles w.r.t each cache set in a structure called cache_trace. Similarly in PRIME+PROBE attack, In order to recover the secret key, we perform the correlation between the intermediate value of the guessed key and every bit of the cache addresses acquired by T-tables. In order

to perform correlation, we may convert our cache traces in binary traces by using a suitable threshold (this is more of an optimization step). We use different leakage models like Bit, ID, and Zero Value models to hypothetically determine the intermediate value. Another most important technique which is utilized to identify how many traces are sufficient in order to recover the full key is *Key Rank Estimation*.

6 Results and Analysis

In this chapter, we discuss and analyze the results of cache attacks strategies on ARM. The focus of our results and analysis is on our test devices (Section: 5.1). We run a pilot attack on x86 first, to reckon the feasibility of our attack and then we escalate it to our test devices (Section: 5.1).

7 Differential Cache Correlation Analysis (CCA) and Leakage Models

Correlation coefficient provides an extremely efficient way to determine linear relationships between data. In our case, we aim to find correlation between the hypothetical intermediate values for every possible key guess to binary cache trace (which contains 1 for a cache miss and 0 for a cache hit on a bunch of addresses) based on certain leakage models. It is to be mentioned that only 4 MSBs of the 16 attacked addresses are important as cache traces operate on cache line level (containing 16 elements each) not on individual address level. We attacked sbox-out of 1^{st} round of AES encryption on Nexus 5 to demonstrate the applicability of cache template attacks. We use our attack strategy along with various leakage models [36] (they are used for power-side channel analysis) to attack AES last round on our test devices(5.1). In this section we will discuss these leakage models and their usability w.r.t our attacks scenarios. We use Inspector (an advanced tool for side-channel analysis) [1] to perform correlation analysis which we term as *Cache Correlation Analysis* (CCA). The non-linear behavior of sbox [36] is key to this analysis. A one bit difference at an sbox input leads to a difference of several bits at the output. Hence, even if a key hypothesis is only wrong in one bit, the output in sbox will be different in several bits. Therefore, while attacking the output of sbox the correlation for all wrong key hypothesis is significantly smaller than the correlation for the correct one. Hence, we mount our correlation analysis on the intermediate results that occur after the sboxes (sbox out) in the first round and before the sboxes in (sbox in) last round.

– **Using ID Leakage Model**

```
ID + 0--------------------N
```

ID leakage model operates on byte level and is suitable for addresses which are close to the either extremes (beginning(0) and ending(N)) of the table. So an address (an index used for lookup) which is either close to 0 or close to N, there will always be a very high correlation. For e.g in the case of address 0 there will be a high correlation when the intermediate value is 0 and if it's higher, the result would be opposite. This is a linear dependency (0 : yes, > 0 : No). The inverse holds true for address N. For all addresses in between, this dependency slowly decreases towards the middle and goes to the negative extreme for N. The reason can be attributed to the fact that the impact of entire byte address correlating with intermediate value is taken into consideration instead of the 4MSBs. Additionally, in such a case neither low values nor high values increase the chance of a hit which subsequently means no linear relationship and no peak. The effect is plotted in figure::

– **Using Bit Leakage Model**

```
BitX + 0--------------------N
```

This model depicts a linear dependency in between the model and the addresses being used during encryption (as shown in sample cache traces) and therefore it is detectable with correlation. It is nice for White Box Cryptosystems since we can circumvent several encoding schemes. This model gives rise to interesting patterns as we take into consideration only a part of the index (address) in the correlation trace.

In order to simplify the assumption lets take into account how 16 cache line addresses would be represented in binary Address0: 0000
Address1: 0001
Address2: 0010
Address3: 0011
Address4: 0100
Address5: 0101
Address6: 0110
Address7: 0111
Address8: 1000
Address9: 1001
Address10: 1010
Address11: 1011
Address12: 1100
Address13: 1101
Address14: 1110
Address15: 1111

Now if we correlate the MSB of our intermediate (which is the index for this table of addresses) we get some interesting results. For example: having the MSB at 1 means that addresses 0-1 and addresses 4-5 and so-on so-forth cannot be used and having the MSB at 0 means that there is a 50% chance

that addresses from 0 to 16 are alternatively used and one can spot a negative correlation at an odd number of addresses.
This effect is shown in (Figure:**??**) for the 4 most significant bits (MSBs) of a 8 bit lookup table (sbox) on x86 architecture (Intel i5) based *DELL* machine . (Figure:**??**) also, demonstrates the effect of 7^{th} MSB of the index value. **This information about the correlation in between the MSB of the address and the bits of intermediate value is of significant importance**. As now, we know for sure that which leakage model depicts our cache trace characteristics in the most accurate way. As bit model gives the bit level granularity (shown in Figure:**??**) and depicts the correlation of each bit with the intermediate value, it is a suitable choice for cache correlation analysis in the later experiments.

8 DCA on ARM

8.1 EVICT+RELOAD

EVICT+RELOAD cache attack is successful in full key recovery on x86 architecture using only 1100 traces as shown in Figure:5. In the following results, we take into consideration only the information related to the 4MSBs of the addresses, as cache operates at cache line level and each line contains only 16 elements of T-tables depicted by 4MSBs of the addresses which contain the useful information. However, for comparison purposes, we also investigate the variation in results when information related to every single address is taken into consideration.

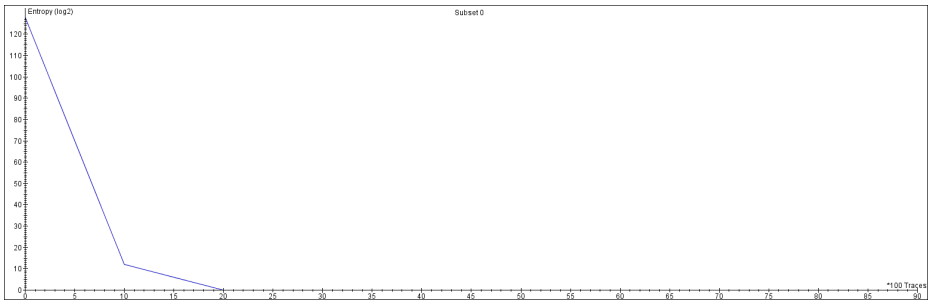


Fig. 5: X86: key rank estimation Using 1100 encryptions

The results of EVICT+RELOAD attack, on OpenSSL 1.0.1g AES implementation vary quite significantly on our target devices. Figure:6 demonstrates that 8000 traces are not sufficient to recover the full key from a crypto process running on *Nexus 5* using key estimation technique [32]. However, the bit entropy is reduced significantly to 14.5 bits. In the case of *Samsung Galaxy S4*, the minimum number of traces required to recover the full key is more than 8000 as can be seen from Figure: 7. It is shown that the key bit entropy gets reduced to

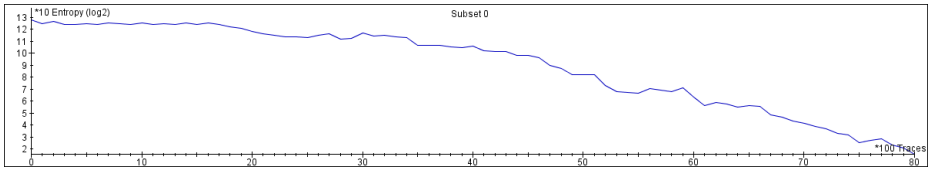


Fig. 6: Nexus 5: key rank estimation using 8000 traces

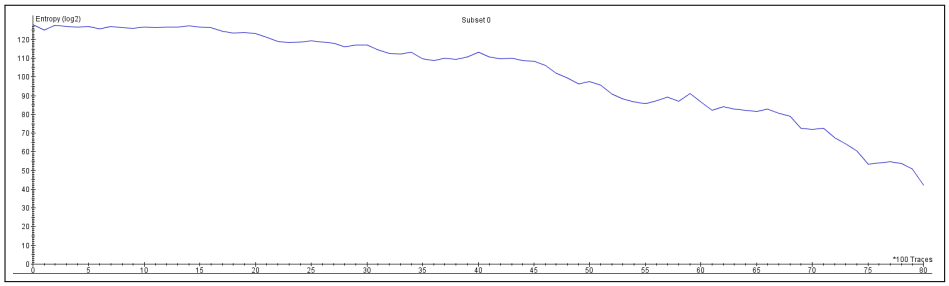
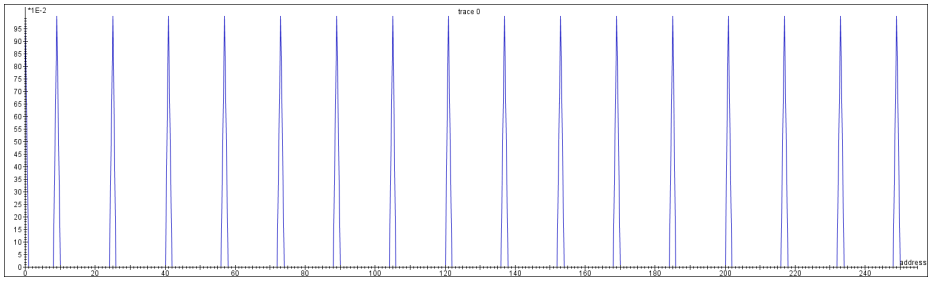
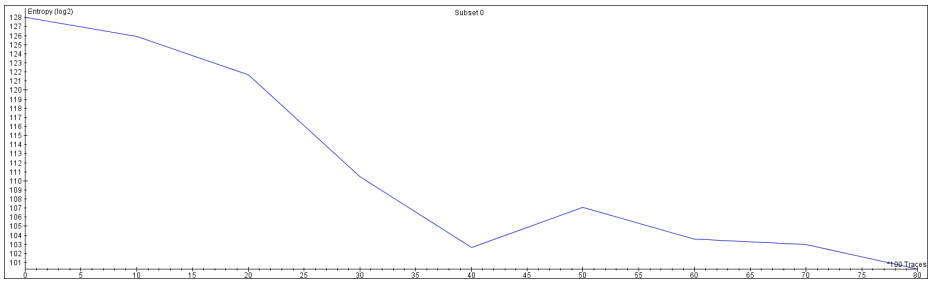


Fig. 7: Samsung Galaxy S4: key rank estimation using 8000 traces

only 48 bits. There is an interesting effect which can be highlighted by using cache hit/miss ratio of every individual address instead of cache line addresses to recover the full key.



(a) Nexus 5: all 256 addresses depicted in cache traces



(b) Nexus 5: key rank estimation using 256 addresses

As, can be observed from Figure:8a, that tables are a little misaligned and not each one of the 16 addresses is present in a cache line, depict the same behavior of the line (due to misalignment). Figure: 8b demonstrates the effect of key rank estimation and correlation analysis on 9000 cache traces.

8.2 PRIME + PROBE

The key estimation results associated with the PRIME+PROBE technique differ from the EVICT+RELOAD because of the following reasons:

- There is no concept of dedicated shared memory involved in between two processes in PRIME+PROBE. Filtering of results on the basis of additional parameters is required.
- Noise is added as some other processes can cause evictions and context-switches and may hamper the results in the cache trace sets.
- It requires more addresses to be monitored in comparison to EVICT+RELOAD and is more time-consuming.

We utilize the same number of traces as used for EVICT+RELOAD technique as a benchmark for PRIME+PROBE technique. Initially, we observed that we are not able to recover the full key on Nexus 5 or Samsung Galaxy S4 with 8000 traces as other processes were running on the smartphones, however, the key entropy reduces to a certain extent. Later, we run only our crypto and victim applications on Nexus 5 and the results are shown in Figure:9.

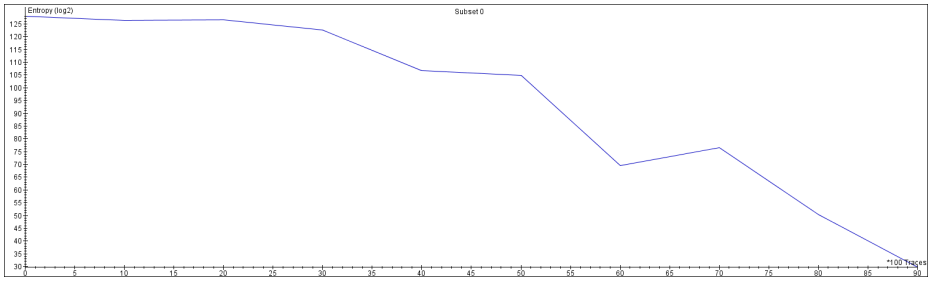
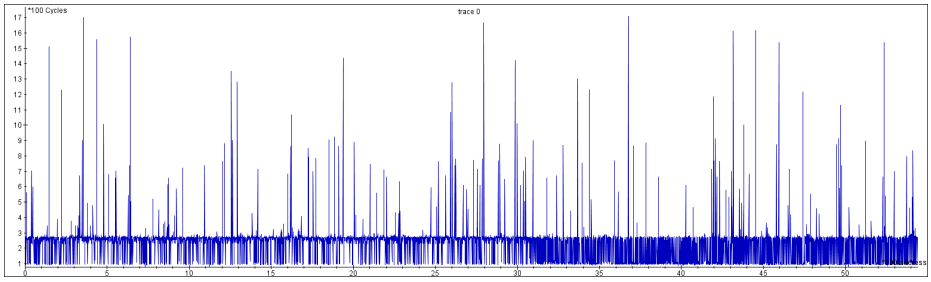


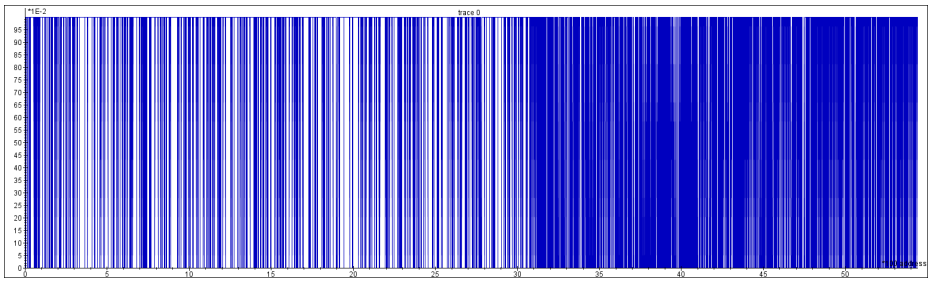
Fig. 9: Nexus 5: Prime+Probe results with only spy and victim process in runnable mode

8.3 White-Box Cryptosystems

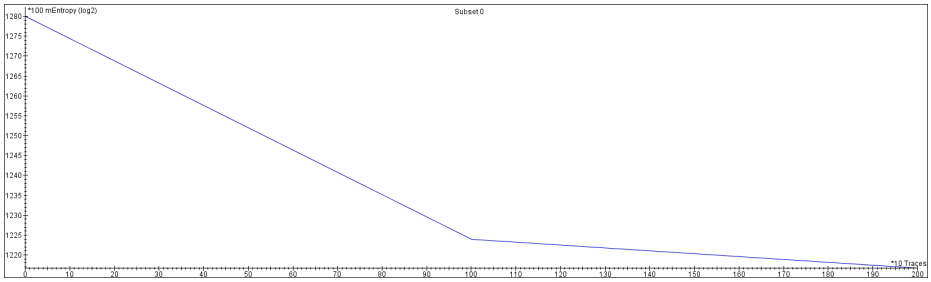
The most important goal of attacking a white-box [2] is to extract the key of the cryptographic algorithm. We implemented an EVICT +RELOAD attack on the crafted traces (Figure: 10). Finally, we find co-relation between the known cache traces and the ideal data. It should be noted that whether a side-channel attack would work or not is very difficult before trying to attack it. We tried our attack on a custom white-box implementation (proposed by Chow et al). Our results



(a) Cache trace-set for access time w.r.t. addresses used by white-box



(b) Binary Threshold applied to cache trace-set



(c) Key Rank Estimation on the newly generated trace-set

Fig. 10: EVICT+RELOAD Cache Attack on a White-Box Cryptosystem

also suggest that there is linear relationship between the number of traces and the key entropy. However,we could not reduce the entropy beyond 120 bits while using only 2000 traces as shown in Figures:(10a),(10b) and (10c). If we increase the number of traces then we should be able to recover the complete secret key. In our work, we confined ourselves to 2000 traces as the trace acquisition alone took more than 30 minutes which is more than the stipulated time for a realistic attack. It can be attributed to the following reasons:

- The tables used in this white-box implementation are large in number and are very small. They could fit into one cache line. This hampers our attack results as our granularity of monitoring is limited to one cache line.
- Secondly, we cannot perform a profile phase as these tables are randomly created at every execution.
- During the generation of the tables in this white-box, it has been ensured that they perfectly align to a cache-line. Hence, as one can easily understand that this white-box was particularly created while keeping cache-attacks in mind.

We tried to attack several academic white-boxes but they were immune to our attack owing to one or another above mentioned reasons. It should be noted that input/output encodings do not have any impact on our results. This finding could assist future attacks on white-box implementations where encodings do annoy the attackers a lot.

9 Impact Analysis of Attack Assumptions

We aim to analyze the impact of various attack assumptions on our results.The assumptions would be analyzed with respect to the success rate (success rate refers to the percentage of successful recovery the secret key during an attack) Following are the results w.r.t various attack parameters:

- **Impact of prefetcher:** While attacking an artificial application when we used de Bruijn cycle, we could avoid the effects of hardware prefetching. On the other hand, when we accessed the array elements in a progression, it triggered the hardware prefetcher as expected.
- **Impact of ASLR:** In case, we do not disable ASLR then the AES t-tables are slightly misaligned. It does not completely affect our results although adds a bit of noise. On the contrary, in some cases it can also be beneficial as one can directly get the key byte as shown by Spreitzer et al. [35].
- **Impact of operating System:** The Linux offerings like *shared memory*, *mmap* and *pagemap* are exploited in our cache attack. Android is based on Linux and as discussed in Chapter-1, Android follows principles of sandboxing etc to protect its application but the mysterious area of shared hardware is not paid attention to. Privileged access to cycle counter and cache flush instructions do make our task a little bit difficult but cache coherency came to our rescue. We could deploy the cross core cache attacks on our test devices 5.1 using cache coherency offered by AMBA ACCI.

- **Impact of cache state before the attack:** Our attack does not work on the cache misses or cache hits unlike previous works [29][10][43] [18]. There can be three different scenarios in which our cache attacks would perform differently as stated below :
  - **Empty initial state:** This is the most favored state for cache attacks as it provides with cold cache misses which were generally the basis for previous attacks on x86. We used this cache state to test our attacks. However, our attack works even without an empty initial state.
  - **Forged initial state:** In these attacks, the attacker should be able to control the initial state of cache as per his/her requirement. This is suggested to manipulate the number of cold start misses. It is equivalent to empty initial state in a way as it also involves flushing of cache followed by some fake encryptions. However, there is a difference as it uses conflict misses instead of cold misses to gain information about the significant key bytes.
  - **Loaded initial state:** As the name suggests, if the tables are already loaded in the memory then that state is called loaded initial state. We utilized this state as well for our attacks and it reduced the number of traces required to recover the complete key.
- **Number of cache traces/encryptions:** The relationship between the number of traces or encryptions required and the success rate of our attack are linearly dependant. If we have more traces, that amounts to more information and eventually, the chances to recover the complete key increases as well.
- **Impact of the privileged and unprivileged mode:** Industry has always argued that if an attacker has an access into victim's execution space then there is no point in carrying out a more complicated low level microarchitectural attack. To answer this, we made our attack work in both privileged and unprivileged modes. However, in a privileged mode we have access to cycle counter, consequently, the success rate increases and the number of encryptions required decreases. In unprivileged mode, we made use of timing mechanisms like *perf* and *monotonic clock*. The results were still promising, however number of required traces increased with the change in the choice of the timing mechanism.
- **Impact of eviction strategy:** It is the most crucial aspect of the attack as an eviction strategy with 100% eviction rate and lower execution time is essential. Using Cache Eviction Strategy Evaluator [26] our task became fairly easy and we could come up with an efficient strategy. In this study, we did not try the eviction strategies suggested by [29][44] as they were time-consuming.
- **Impact of the type of attack:** Prime+Probe is a much more realistic attack but we could not completely recover the key using Correlation Analysis, given the same number of traces used for EVICT+RELOAD. On the other hand, Evict+Reload in various attack scenarios results in a successful key recovery as discussed in previous sections.

10 Conclusion

We present a novel evaluation strategy for cache side channel attacks on Android smartphones(Section: 5.1). Our study discusses the applicability of cache attacks (PRIME+PROBE, EVICT+RELOAD) on a victim application making use of native code on our test devices (5.1). It emphasizes the effect of the cache intial state, timing probe, compiler, access settings, operating system etc on the generated cache traces. Further, we discuss the number of key bytes recovered by analyzing the cache behavior. Our experiments are not entirely dependant on the above-mentioned specifications and hence can be reproduced with much ease. We perform cache attacks on OpenSSL 1.0.1g and OpenSSL 0.9.7a AES implementation on our test devices 5.1 using a state-of-the-art attack strategy based on the works of Spreitzer et al. [35] and Tromer et al. [29]. In the previous works [29][6][16] picture analysis was fundamentally used to identify the key candidates. Correlation Analysis was used in timing attacks but not in access driven cache attacks on ARM. Our results establish the successful implementation of statistical techniques like *Correlation Analysis and Key Rank Estimation* on cache trace-sets for recovery of the complete secret key of AES. Using the proposed method, we quantitatively evaluate the transition with the increase of the number of plaintexts. We also discuss the performance of our attack in varying attack scenarios. Finally, we demonstrate the cache attack on a vulnerable implementation of AES resembling a white-box implementation and then test its applicability on a custom white box implementation of AES.

Future work related to the investigation of cache attacks on white-box cryptosystems seems promising. We believe that incorporation of statistical analysis with access-driven cache attack opens door to a major research in the field of white-box cryptosystems. Furthermore, launching these cache attacks without any privilege escalation or assistance from performance counters will ease the implementation. The possibilities of new cache attacks and new attack vectors are also a part of future work. Different countermeasures to such statistical analysis could be researched. For instances, techniques like code obfuscation can be implemented but the level to which it should be done remains questionable.

References

1. Inspector sca. `https://www.riscure.com/security-tools/inspector-sca/`.
2. Side channel attack against white-box cryptography on android. `https://github.com/edermi/papers/blob/master/Side-channel%20attacks%20against%20whitebox%20cryptography%20on%20Android/thesis.pdf`.
3. These are not your grand daddy's cpu performance counters- cpu hardware performance counters for security. `https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters`.
4. Smartphone os market share, 2016 q3. `http://www.idc.com/promo/smartphone-market-share/os`, 2016.
5. Ali Can Atici, Cemal Yilmaz, and Erkay Savas. Remote cache-timing attack without learning phase. *IACR Cryptology ePrint Archive*, 2016:2, 2016.

6. Daniel J Bernstein. Cache-timing attacks on aes.
7. Andrey Bogdanov, Thomas Eisenbarth, Christof Paar, and Malte Wienecke. Differential cache-collision timing attacks on aes with applications to embedded cpus. Springer.
8. Joseph Bonneau and Ilya Mironov. *Cache-Collision Timing Attacks Against AES*, pages 201–215. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
9. Billy Bob Brumley. *Cache Storage Attacks*, pages 22–34. Springer International Publishing, Cham, 2015.
10. Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *ASIACRYPT*, 2009.
11. Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.
12. Joan Daemen and Vincent Rijmen. The design of rijndael:aes the advanced. *Journal of Cryptology*, 4(1):3–72, 1991.
13. Catherine H. Gebotys and Brian A. White. A sliding window phase-only correlation method for side-channel alignment in a smartphone. *ACM Trans. Embed. Comput. Syst.*, 14(4):80:1–80:22, September 2015.
14. Cezary Glowacz, Vincent Grosso, Romain Poussier, Joachim Schueth, and François-Xavier Standaert. Simpler and more efficient rank estimation for side-channel security assessment. In *International Workshop on Fast Software Encryption*, pages 117–129. Springer, 2015.
15. Ben Gras and Kaveh Razavi. Aslr on the line: Practical cache attacks on the mmu. 2017.
16. Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379. ACM, 2016.
17. Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016.
18. Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 897–912, Berkeley, CA, USA, 2015. USENIX Association.
19. 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.
20. Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205. IEEE, 2013.
21. John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. *Computer SecurityESORICS 98*, pages 97–110, 1998.
22. Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud.
23. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in cryptologyCRYPTO99*, pages 789–789. Springer, 1999.
24. Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

25. Cédric Lauradoux. Collision attacks on processors with cache and countermeasures.
26. Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, 2016. USENIX Association.
27. Daneil Gruss Moritz Lipp. `https://github.com/IAIK/armageddon/tree/master/libflush`.
28. Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A refined look at bernstein's aes side-channel analysis. In *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '06, pages 369–369, New York, NY, USA, 2006. ACM.
29. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.
30. Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel.
31. Colin Percival. Cache missing for fun and profit.
32. Romain Poussier, Franois-Xavier Standaert, and Vincent Grosso. Simple key enumeration (and rank estimation) using histograms: An integrated approach. In *CHES*, pages 61–81. Springer, 2016.
33. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.
34. J. M. Smith and A. B. Jones. Ecdsa key extraction from mobile devices via nonintrusive physical side channels. 2016.
35. Raphael Spreitzer and Thomas Plos. *Cache-Access Pattern Attack on Disaligned AES T-Tables*, pages 200–214. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
36. Thomas Popp Stefan Mangard, Elisabeth Oswald. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. 1 edition, 2007.
37. Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of des implemented on computers with cache. Springer.
38. Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1675–1689. ACM, 2016.
39. Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on aes in virtualization environments. In *International Conference on Financial Cryptography and Data Security*, pages 314–328. Springer, 2012.
40. Wikipedia. Cache coherence — wikipedia, the free encyclopedia, 2017.
41. Wikipedia. Cpu cache — wikipedia, the free encyclopedia, 2017.
42. Zhao Xinjie, Wang Tao, Mi Dong, Zheng Yuanyuan, and Lun Zhaoyang. Robust first two rounds access driven cache timing attack on aes. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 3, pages 785–788. IEEE, 2008.
43. Yuval Yarom and Katrina E Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, volume 2014, pages 719–732, 2014.
44. YongBin Zhou and DengGuo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing.