NTNU – Trondheim
Norwegian University of
Science and Technology

Exploring Instruction Cache Analysis

On Arm

Stian Valentin Svedenborg

Master of Science in Physics and Mathematics
Submission date: September 2014
Supervisor: Harald Hanche-Olsen, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Abstract

This thesis explores the challenges of implementing an instruction cache side-channel attack on an ARM platform. The information leakage through the instruction cache is formally discussed using information theoretic metrics. A successful Prime+Probe instruction cache side-channel attack against RSA is presented, recovering 967/1024 secret key bits by observing a single decryption using a synchronous spy process. Furthermore, an unsuccessful attempt is made at decoupling the spy from the victim. Finally, the current state of countermeasures against software based cache side-channel attacks are summarised.

Contents

List of Figures

Chapter 1

Introduction

Information security has been a hot topic for as long as people have had secrets. Over time, techniques to share these secrets securely have evolved from simple riddles into rigorous mathematical systems with proofs and bounds on their security.

For many years there has been an unending struggle between the designers of crypto-systems and the people who want to break them. The crypto-community's goal is to always stay one step ahead of the attackers. They do so by actively breaking each others' systems, publishing the results and, most importantly, suggesting ways to defend against the attacks they find.

In more recent years, as crypto-systems become more robust against classical cryptanalysis, researchers have been looking at alternative ways to break and improve the security of crypto-systems by looking outside the pure mathematics of the system and into their actual implementation in real world usage, this is side-channel analysis.

1.1 Side-Channels

In cryptology, a side-channel is defined as unintentional leakage of information through a secondary channel. It is important to understand that side-channels can be completely separate from the mathematical security of the crypto-system. Even provably secure systems may be susceptible to side-channel analysis, owning to the fact that side-channel analysis usually attack the implementation of the system, and not only the mathematics upon which the system's security is based.

To give an example, most modern crypto-systems are implemented on computers. When executed on a computer, calculations take time, processors use power and memory changes state. This information may in turn be used to infer certain properties of the plain-text or even actual bits of the key of a crypto-system as it is run. Because of this, side-channels and related attacks have become a popular topic of study over the last few decades.

One of the earlier results in side-channel analysis is the attack by Bleichenbacher in 1998 [13]. In this attack an arbitrary message of RSA Laboratories' PKCS#1 v1.5 is decrypted by the use of about 1000000 queries to a padding oracle. Bleichenbacher showed that this oracle could be created by observing how the receiver rejects a chosen cipher text. The attack works by observing the error-messages or even simply the delay of a decryption operation and uses this information to piece together the unencrypted message. Bleichenbacher's attack is a side-channel attack because it uses the details about how the decryption process is *implemented* to break the system.

Another example is a template attack on the Digital Signature Algorithm (DSA) by Howgrave and Smart [26]. In their article, they show that if it is possible to extract some bits of information about the nonce used by the DSA, it will be possible to extract the complete key within reasonable time. Howgrave's article does not show *how* one can extract such information, but more recent work has successfully implemented the attack using side-channel analysis [2].

1.1.1 Micro-Architectural Side-Channels

A special class of side-channels is based on the micro-architecture of the computers on which the crypto-systems are run. These kinds of attacks analyse and exploit the way a computer's CPU, cache and other hardware work together to execute a crypto-system.

The main focus of this thesis will be on micro-architectural (MA) side-channel attacks that use the CPU's Level 1 instruction cache to gain information about the crypto-system, introduced below.

Many classes of MA side-channels are already known and this is an active field of study, see for instance [1, 2, 3, 15, 39]. The main classes of known micro-architectural side-channels are outlined below.

Timing Analysis

Pure timing attacks are among the simplest side-channels to understand. They work by measuring the time it takes to complete certain parts of a crypto-system, for which the computational time depends on the value of the secret input, usually the secret key. Kocher [27] showed how this information could be used to extract information from common implementations of RSA, the Diffie–Hellman key exchange protocol and the Digital Signature Standard, and the results may easily be adapted for other systems.

The existence of this side-channel forces implementers to ensure that the timing profiles of sensitive code are either fixed or sufficiently random.

Data Cache Analysis

Data cache attacks [9, 15] apply another strategy to infer information about the secret keys of a crypto-system, namely by observing how the CPU's data cache operates when the CPU is executing sensitive code.

Three kinds of data cache attacks exist: Trace-driven attacks rely on being able to determine whether any given memory access is a cache hit or cache miss[1] as the process is executing. This information may then be used to infer parts of the secret key based on, for instance, the table look-up patterns in some software implementations of AES, cf. [6].

Timing based data cache attacks rely on timing profiles instead of a cache trace. An attacker measures the total time of an operation and makes inferences as to how many cache hits or misses occur during the operation. This kind of attack is based on statistical inference and thus usually requires a much larger sample than a trace-driven attack, but it may in turn be used for remote attacks over a network. Acıiçmez et al. do exactly this in [9] where they use a data cache timing attack to remotely break a popular implementation of AES.

The third type of data cache attacks has been named access-driven and works by observing which cache sets (see section 2.1) the crypto-application uses, but does not care about the exact order as is done in a trace-driven attack. Armed with this knowledge, the attacker can infer which elements in the look-up tables are accessed by the cipher and from that information deduce parts of the private input.

Branch Prediction Analysis

Branch prediction analysis (BPA) [3, 4, 5] is another MA side-channel that was introduced by Acıiçmez et al. in 2006 and is based on the Branch Prediction Unit (BPU) of the processor. The BPU is a piece of hardware that tries to predict ahead of time where the execution of a program will branch off to. That is, whether a conditional statement will evaluate to true or false. This kind of prediction is needed because modern processors often use a technique called pipe-lining, which is outside the scope of this thesis. Let us suffice to say that a failure to correctly predict a branch will cause execution to stall while the pipeline recovers.

There are, as with cache attacks, several different attack strategies in BPA. The most noteworthy is Simple BPA (SBPA) [5] which attempts to discover the result of any and all branches in the attacked code. When it was introduced in late 2006, it was shown that SBPA could potentially be used to extract the complete private key from simple implementations of RSA in one iteration.

[1]Cache architecture and terms like *cache hit* or *cache miss* are defined in section 2.1.

The attack in [5] utilises a spy process that continuously forces the BPU to expect that *no* branches will be taken. Thus, whenever a branch *is* taken the process stalls while the pipeline recovers. It is this delay that is measured and ultimately used to infer the private input.

Instruction Cache Analysis

The instruction cache (I-cache) side-channel is a natural continuation of SBPA, as both are concerned with analysing the execution flow of the crypto-application. However, whereas SBPA is concerned with the predictions of the BPU, I-cache attacks analyse how code is being loaded into the CPU's instruction cache.

Instruction cache attacks are based on a simple and powerful assumption: Any two distinct pieces of code will likely be loaded into different parts of the instruction cache. This assumption, which is often true in practice, allows us to follow the execution flow of the application as it is run.

In short, an I-cache attack utilises a spy-process that runs parallel to the attacked code *on the same physical CPU core*. This spy process monitors the instruction cache and notes whenever certain patterns of code are loaded into the cache. Based on this information it is possible to infer which code of the crypto-process was actually run. Together with a careful analysis of the crypto-process, this information may be used to piece together the secret key, or other sensitive information.

An important point to mention is that both SBPA and I-cache analysis attack code where execution flow is dependent on the input to the system. Unfortunately, this is often the case: Popular implementations of RSA, DSA and more have been known to have key-dependent flow, cf. [2, 4, 5, 8]. Any implementation of a system in which the execution flow is dependent on secret state may be vulnerable to these attacks.

Shared Functional Units

The final known micro-architectural side-channel arises due to the fact that on some systems, where two threads may run simultaneously on a single processor core[2], the two threads must share the use of certain circuits. For instance, if simultaneous threads share a integer multiplier unit this may create a side-channel [7, 10]. As processor manufacturers make more optimizations in terms of area to concurrency, it is likely that we will see more side-channels of this kind.

An attack using these functional units will attempt to follow the instruction flow of the algorithm in much the same way as SBPA and I-cache analysis.

[2]This is called Simultaneous Multi-Threading (SMT) and is discussed further in section 2.4.

1.2 Thesis Statement

In this master thesis we will try to build on the work by Acıiçmez and others and attempt to implement an *instruction cache side-channel attack* on an ARM system. With exception to the work of Köpf et al. in [32], there has been little published research regarding micro-architectural side-channel analysis on ARM platforms, as much of the published work rely directly or indirectly on instructions and technology that is not as easily accessible on ARM platforms.

It is the intention of this thesis to explore the potency and difficulties of implementing an instruction cache side-channel on ARM platforms.

1.3 Thesis Structure

In chapter 2 we will cover the necessary background from previous work and give a short introduction to cache architecture.

Next, chapter 3 will cover the framework we will use to analyse the information leakage in the side-channels we will explore.

Chapters 4–6 will contain our main results, in which we delve into the practical analysis of several side-channels. Here, we present both positive results chapters 4 and 5 and a negative result in chapter 6.

Finally, in chapter 7 we recite the current state of countermeasures to protect against cache based side-channels before we summarise and conclude our results in chapter 8.

Chapter 2

Background Material

In this chapter we will explain in further detail what a cache is and how we are able to use the instruction cache (I-cache) as a side channel.

2.1 Cache Architecture

In computer architecture, a cache is a fast memory component that is located between the processor(s) (CPU) and the main memory (RAM). The need for a cache has arisen from an increasing inequality in the relative speed of processors and memory technology. In short, as processors have become faster and faster, memory technology has fallen behind. The solution to this problem has been to place a small portion of very fast (expensive) memory between the processor and the main memory to temporarily store frequently used data. This is the cache. A high level layout of CPU memory architecture is shown in figure 2.1.

On modern computers, there are generally 2–3 levels of cache. Level 1 (L1) is the smallest and fastest and usually split into the L1-I and L1-D cache[1], one for instructions and one for data. Level 2 (L2) is slightly slower and larger and, on some platforms, Level 3 (L3) offers yet another layer that is bigger and slower.

The idea is that when the CPU needs to access the memory, it will first check if the required data is in the L1 cache, then check L2, then L3. If the data is not found on either level, it is loaded into the caches from the main memory. Thus, if the same data needs to be accessed again it will be retrieved much faster.

The exact architecture for the caches of processors are as diverse as the processor manufacturers and models themselves. There are, however, some similarities

[1]What we are describing is often referred to as a *Harvard architecture*. It allows the CPU to access instructions and data along separate paths. In contrast, a *von Neumann architecture* has only one data path shared between instructions and data.

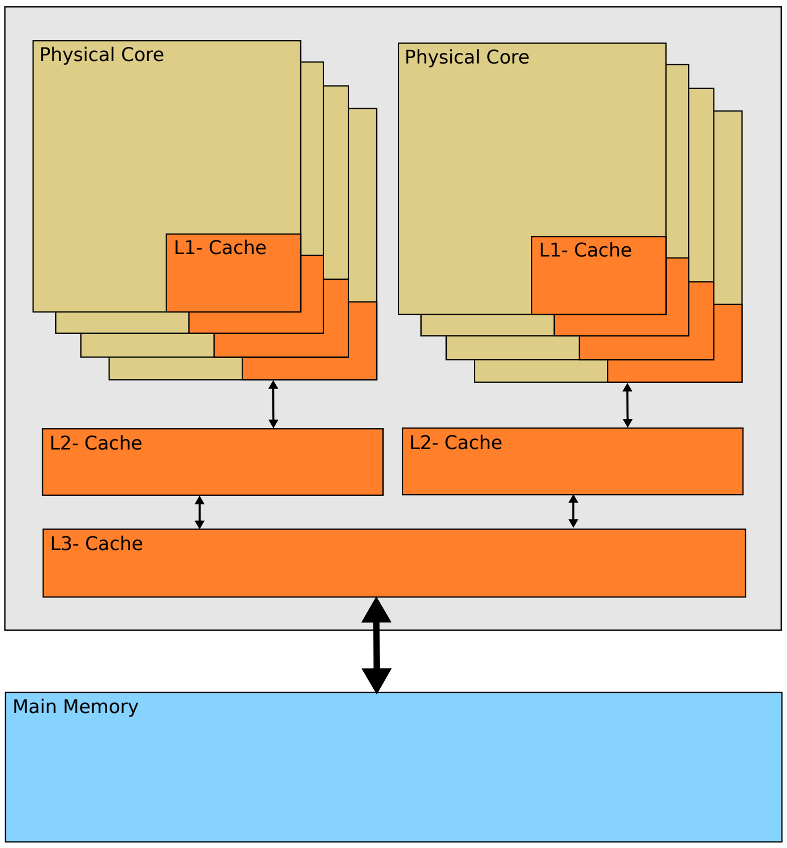


Figure 2.1: Typical CPU memory layout. Each physical core has its own L1 cache. A group of cores typically share an L2 cache. Multiple core groups share the L3 cache (if present).

and key terms which will be defined in this section. Most of the material in this section is adapted from [20].

Definition 1. The atomic amount of memory that may be loaded into the cache is called a *cache line.*

If you are to access a 32-bit integer that is not currently in the cache, the cache will load one or more cache lines such that all 4 bytes of the integer are now residing in the data cache. This is done completely analogously for the instruction cache. If you are to execute a piece of code not currently in the cache, a multiple of full cache lines will need to be loaded into the cache.

Definition 2. A memory access is called a *cache hit* if the memory to be loaded is already residing in the cache. An access that is not a cache hit is called a *cache miss.*

Definition 3. A memory object (logical data stored in memory) whose size would fit into N cache lines is called *cache unaligned* if an access to the object would cause $N + 1$ cache lines to be loaded into the cache. A memory location that is not cache unaligned is called *cache aligned.*

Again looking at the same integer as above. If the memory location of this integer is aligned such that the first two and last two bytes reside in different cache lines, then both cache lines will need to be loaded into the cache before the CPU can access the memory.

Definition 4. The *set associativity* of the cache is a measure of how many valid cache lines any one memory location may be loaded into. This set of cache lines is called a *cache set.* In a *fully associative* cache, any memory location may be loaded into any cache line. In an *N-way associative* cache, a memory location may be mapped into one of exactly N fixed locations. A 1-way associative cache is called *directly mapped.*

Definition 5. When a cache line is loaded into a cache set that is full, another cache line has to be *evicted* to a higher level memory. The algorithm used to choose which cache line to evict is called the *replacement policy.* The most commonly used policy is to evict the *Least Recently Used* (LRU) cache line.

2.2 The Instruction Cache

In the instruction cache the memory containing the machine code to be executed is loaded. Conversely, the data to be processed is loaded into the data cache. The

Cache size:	32KB
Associativity:	2-way
Cache line size:	Fixed, 64 bytes
Replacement policy:	Least Recently Used (LRU)

Table 2.1: Example cache features from an ARM CORTEX A15MP, L1 instruction cache.

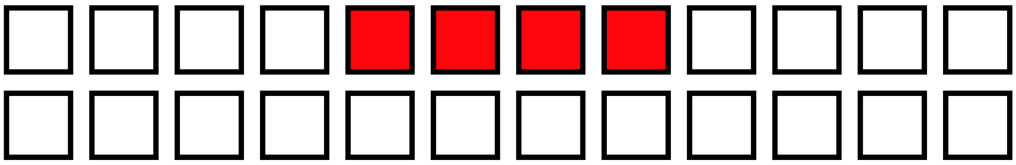
reason for this split is the locality property of both code and data. In general, computer programs access both the instructions and the data they process such that recently used memory will probably be used again in the near future. Program code shows an even stronger locality than data due to loop structures, functions etc. Thus, caching the instructions from slower memory will, in general, result in much faster execution. Furthermore, relatively few instructions may operate on large amounts of data. In this case, the separation of the L1 caches may drastically improve performance as the instructions will not need to be fetched repeatedly from higher order memory.

To give an example: Say we are to run the code given in listing 1. How would this code be loaded into the cache? Notice the functions `multiply` and `square`. These functions will probably be mapped into different cache sets when loaded into the cache. Now, if we assume a cache with 24 cache sets, the corresponding mapping could turn out like shown in figure 2.2 on the following page. Keep in mind that the cache sets could just as easily overlap.

```
def pow( base, exponent )
    multiplier = base
    result = 1
    for bit in exponent:
        if bit == 1:
            result = multiply(result, multiplier)
        multiplier = square(multiplier)

    return result;
```

Listing 1: Pseudocode for a simple square and multiply exponentiation algorithm.


(a) `multiply`

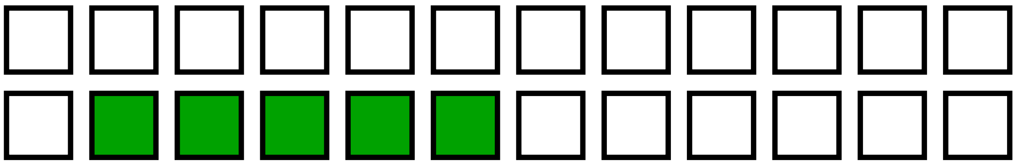

(b) `square`

Figure 2.2: Example mapping of the functions `multiply` and `square` into the corresponding cache sets in an I-cache with 24 cache sets.

2.3 The Spy Process

The basic idea of an I-cache trace attack is that the instruction cache is shared between the different execution contexts. This is a property that can be exploited. As shown in [39] it is possible to use cache access patterns to send information across security borders, and it is a similar idea that is the basis of all cache attacks: Information about the crypto-system will leak through the cache access patterns.

The spy process in figure 2.3 on the next page is taken from [2] and shows a spy process monitoring the instruction cache. It works by first aligning the code to that of one cache set. It is then executed in such a way that it will fill one cache set with inert code and measure how long it takes to fill the entire set, using the `rdtsc` instruction on x86 platforms. Finally, it repeats the procedure for all cache sets. An attack utilising this kind of cache spy is referred to as a PRIME+PROBE attack in the literature.

The idea is that, if no code (other than the spy itself) has accessed a particular cache set, then the instructions stored in that set will be loaded and executed quickly. If, however, another process has accessed that particular cache set in the meantime, the spy's instructions need to be fetched from a higher level memory and the code will execute significantly slower. This difference is measurable and it is this difference that is the physical observable in the instruction cache side-channel.

2.4 Simultaneous Multi-Threading

Another point that must be mentioned is the concept of Simultaneous Multi-Threading (SMT). SMT is a technique employed by some processor manufacturers[2] to create more logical processor cores on one physical core. They do this by replicating only some vital components of the processor and share the remaining components among these logical cores, cf. figure 2.4 on page 13.

SMT is important to instruction cache analysis because the instruction cache is amongst the components shared between the logical cores in an SMT context. Thus it is possible to run the spy-process truly concurrently to the crypto-process, and theoretically we are able to trace the execution flow as it happens.

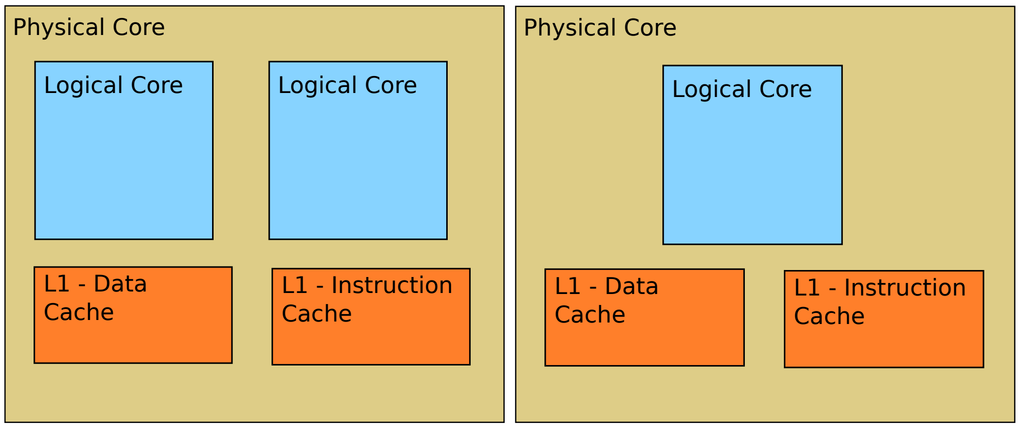
The alternative is to execute the attack in a simulated multi-threading environment (Non-SMT) where each thread is scheduled to run in turn on one physical core. This second approach, although possible, will incur more noise from thread context switching.

Most previous work has been concerned with I-cache analysis in an SMT environment as this greatly simplifies decoupling of the spy from the victim. In this

[2]Intel markets its SMT technology under the name Hyper-Threading.

```
xor %edi , %edi
mov <buffer addr>, %ecx
rdtsc
mov %eax, %esi
jmp L0
.align 4096
L0:
    jmp L64
    .rept 59
    nop
    .endr
L1:
    jmp L65
    .rept 59
    nop
    .endr
...
L64:
    jmp L128
    .rept 59
    nop
    .endr
...
 L448:
    rdtsc
    sub %esi , %eax
    movb %al , (%ecx,%edi)
    add %eax, %esi
    inc %edi
    jmp L1
    .rept 49
    nop
    .endr
...
L511:
    rdtsc
    sub %esi , %eax
    movb %al , (%ecx,%edi)
    add %eax, %esi
    inc %edi
    cmp <buffer len>, %edi
    jge END
    jmp L0
```

Figure 2.3: The instruction cache spy-process for x86 platforms from [2]. The same spy for ARMv7 may be inspected in appendix B.



(a) SMT (b) Non-SMT

Figure 2.4: Comparison of SMT vs. non-SMT processors. On an SMT processor, two or more logical cores share resources.

thesis we will only look at a non-SMT environment which makes the decoupling of the spy significantly harder, cf. chapter 6.

2.5 Data Analysis

In this section we present the necessary practical tools needed to analyse the timing measurements of the spy presented in section 2.3.

2.5.1 Vector Quantization

Vector quantization (VQ) [23] is the process of mapping an N-dimensional vector space down to a finite subset of vectors in the same space, preferably represented as compactly as possible. It is a very potent preliminary screening technique when looking for patterns in the presence of noise. Previous work has shown that this can be very useful when analysing the timing data from the spy-process.

Definition 6. A *vector quantization* is a mapping $Q : \mathbb{R}^N \to \mathcal{C}$ where $\mathcal{C}$ is a finite set of points from $\mathbb{R}^N$. Thus $\mathcal{C} \subset \mathbb{R}^N$. The set $\mathcal{C}$ is often called a codebook.

The idea behind vector quantization is illustrated in figure 2.5 for the two dimensional case, but the same structure is valid for higher dimensions.

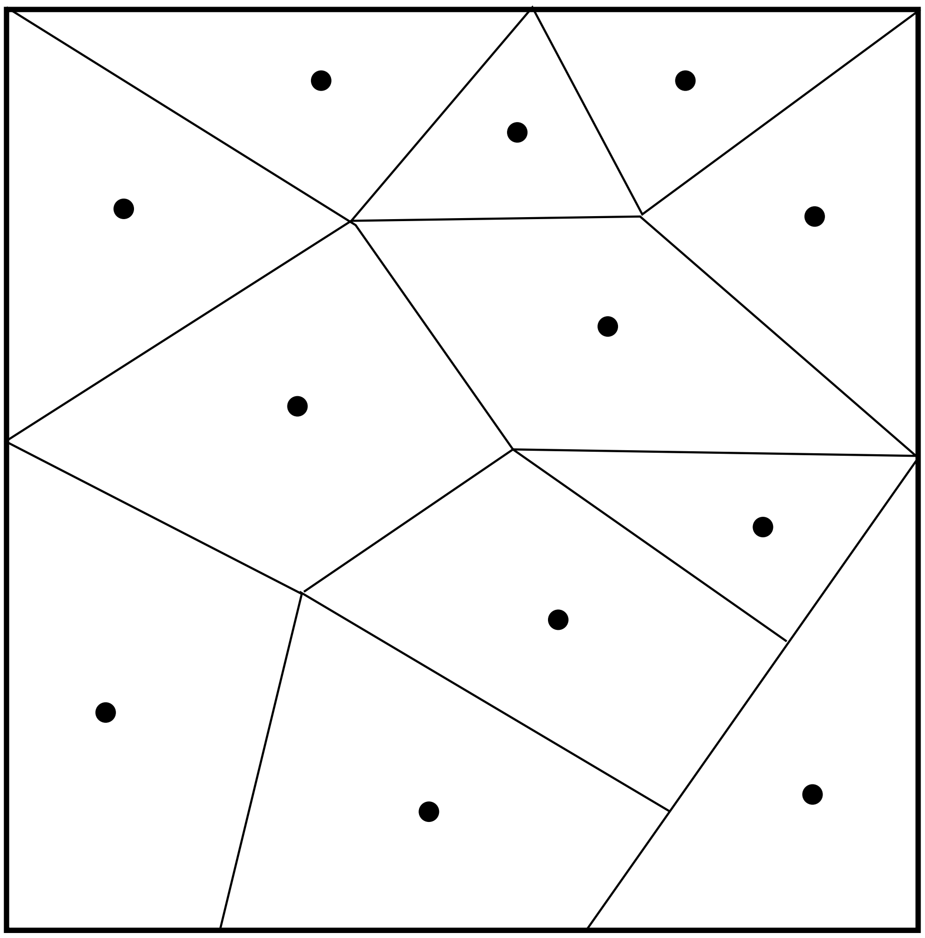


Figure 2.5: Example of a vector quantization of $\mathbb{R}^2$. The lines partition the vector space and the dots mark the representative in the codebook for each partition.

Nearest Neighbour Quantizers

The special class of vector quantizers we are concerned with is called *nearest neighbour quantizers* or Voronoi quantizers. What identifies a nearest neighbour quantizer is that the partitioning of $\mathbb{R}^N$ is uniquely determined by the codebook $\mathcal{C}$ and the use of Euclidean distance in $\mathbb{R}^N$:

$$d(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|_2 = \sqrt{(\boldsymbol{x} - \boldsymbol{y}) \cdot (\boldsymbol{x} - \boldsymbol{y})}. \tag{2.1}$$

Encoding any vector $\boldsymbol{x}$ now becomes a search through $\mathcal{C}$

$$\mathrm{Enc}(\boldsymbol{x}) = \underset{\boldsymbol{y_i} \in \mathcal{C}}{\arg\min}\ d(\boldsymbol{x}, \boldsymbol{y_i}). \tag{2.2}$$

If $\boldsymbol{x}$ is equidistant from two different codewords $\boldsymbol{y_i}, \boldsymbol{y_j} \in \mathcal{C}$ we will simply pick the one with the smallest subscript, thereby enforcing a well defined map.

2.5.2 Self-Organising Map

In the previous section it was shown how to encode a vector using nearest neighbour encoding given a predetermined codebook $\mathcal{C}$. The next problem to address will be how to create such a VQ that best describes a set of training data.

This paper will look at a technique frequently used in signal processing and pattern recognition, namely using a Self-Organising Map[3] (SOM) together with use of Learning Vector Quantization (LVQ) [16, 28, 29].

The idea behind a SOM is to iteratively improve the VQ based on the training data.
Two key properties of a SOM are worth mentioning:

1. The algorithm is randomised, thus subsequent executions may yield different VQs.

2. Codewords that map domains with "similar properties" will tend to be close together in the topology of the SOM. This is of special interest in pattern recognition.

Training a SOM

Before the training can commence, two parameters must be chosen: The number of nodes and a topological structure describing their relationship. Typical structures include: A one dimensional string of beads (each node being a bead) or

[3]Also known as a Kohonen Map after its creator.

various configurations of N-dimensional grids. The most common choice is a two-dimensional hex-grid (also known as a honeycomb structure), as this configuration gives each node 6 equidistant neighbours whilst keeping to only two dimensions. If one is not concerned with grouping similar data together, the layout of the SOM is unimportant.

Once the topological structure is chosen, a metric $\tilde{d}(i,j)$ measuring the distance between nodes i, j in this space is also needed. We use this to define the neighbourhood of node i.

Definition 7. The *R-neighbourhood* of a node i are all nodes j such that:

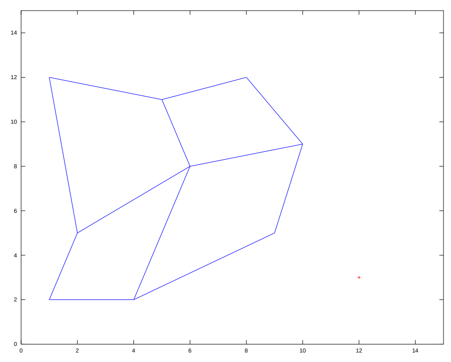
$$N_R(i) = \{j \mid \tilde{d}(i,j) < R\}. \tag{2.3}$$

We further require that

$$R \leq R' \Rightarrow N_R(i) \subseteq N_{R'}(i) \tag{2.4}$$

and

$$i \subseteq N_R(i) \quad \forall i, R. \tag{2.5}$$


(a) Before iteration

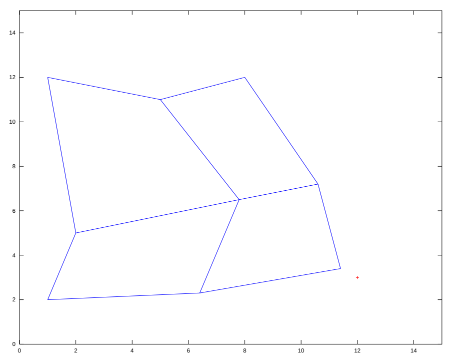

(b) After iteration

Figure 2.6: Illustration of a SOM training iteration. After the iteration, the closest codeword, along with its neighbours are pulled closer to the training vector. This iteration's training vector is shown in red, whilst the internal nodes of the SOM are shown in blue.

The algorithm starts out by assigning to each node a randomly chosen codeword $\boldsymbol{\mu}_i$. Then for a number of iterations, a randomly drawn training vector $\boldsymbol{x}$ is chosen, the node with the closest vector $\boldsymbol{\mu}_i$ is found and all vectors in the R-neighbourhood of i are pulled towards $\boldsymbol{x}$. This process is illustrated in figure 2.6 on the preceding page. During the initial phase of the training, the neighbourhood radius R is slowly decreased until $N_R(i) = \{i\}$. The training may still continue after this point, but only the closest node i will be affected. It is the additional pull on the neighbours of i during the initial phase that gives a SOM property 2 as stated above.

We will now introduce the algorithm for training a SOM.

Algorithm 1. Training a SOM.

Given a training set $T \in \mathbb{R}^M$, the number of iterations N and the SOM configuration (including the number of nodes n and topological structure of the nodes expressed by $N_R(i)$).

Furthermore, let $\alpha(t)$, $\beta(r)$ be two monotonically decreasing functions with range $\langle 0.0, 1.0]$. Likewise, let $R(t)$ be a monotonically decreasing function for the neighbourhood radius.

1. For all nodes j assign a random $\boldsymbol{\mu}_j$ from T.

$$\boldsymbol{\mu}_i \overset{r}{\leftarrow} T.$$

2. Set the time t to zero.

$$t \leftarrow 0.$$

3. While $t < N$, repeat

   (a) Draw a random training vector.

$$\boldsymbol{x} \overset{r}{\leftarrow} T.$$

   (b) Find the node i with $\boldsymbol{\mu}_i$ closest to $\boldsymbol{x}$.

$$i \leftarrow \arg\min_j \left(d(\boldsymbol{x}, \boldsymbol{\mu}_j)\right)$$

   (c) For all nodes j

$$\mu_j \leftarrow \begin{cases} \boldsymbol{\mu}_j + \beta(\tilde{d}(i,j)) \cdot \alpha(t) \cdot (\boldsymbol{x} - \boldsymbol{\mu}_j) & \text{if } j \in N_{R(t)}(i) \\ \boldsymbol{\mu}_j & \textit{otherwise.} \end{cases}$$

   (d) Step the time.

$$t \leftarrow t + 1$$

The resulting SOM is a codebook $\mathcal{C}$ mapping vectors from $\mathbb{R}^M$ to the set of integers $\{0, 1, \ldots, n-1\}$ with a corresponding value $\boldsymbol{\mu}_i \in \mathbb{R}^M$. For our use case we wish to label each node in the SOM according to a majority vote amongst the (labelled) training vectors.

2.5.3 Learning Vector Quantization

Learning vector quantization (LVQ) is a technique building on the idea of a SOM. The idea is to further increase the classification power of the vector quantizer reducing the number of "border cases".

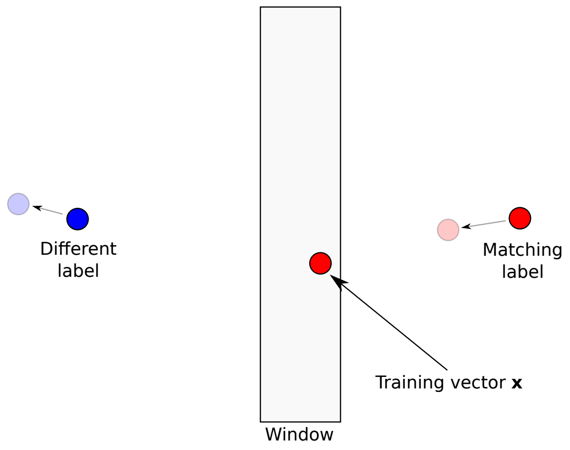


Figure 2.7: Learning step of LVQ3. Assuming the two closest codewords have different labels. The codeword with a label matching that of $\boldsymbol{x}$ is pulled closer to $\boldsymbol{x}$ whilst the codeword with a different label is pushed away. (If $\boldsymbol{x}$ is in the window.)

The main idea is illustrated in figure 2.7. In short, a labelled training vector $\boldsymbol{x}$ is randomly chosen from the training data T. Next, find the two closest labelled codewords $\boldsymbol{\mu}_i$, $\boldsymbol{\mu}_j$ in the vector quantizer $\mathcal{C}$. Once the two codewords are chosen we check whether the label of $\boldsymbol{x}$ match the label of either $\boldsymbol{\mu}_i$ or $\boldsymbol{\mu}_j$. If neither match, discard $\boldsymbol{x}$ and draw a new training vector.

Otherwise, check if $\boldsymbol{x}$ is "in the window", that is, define a decision border between the codewords $\boldsymbol{\mu}_i$ and $\boldsymbol{\mu}_j$ and state that if $\boldsymbol{x}$ is in this window, it is a border case and we continue, otherwise we discard it and draw a new training vector.

Now, if exactly one of the labels match, we pull the codeword with the correct label closer to $\boldsymbol{x}$ and push the other away. Otherwise, both labels match and we pull both closer to $\boldsymbol{x}$, albeit a little shorter (reduced by a factor ϵ).

This technique is the third iteration of LVQ as presented by Kohonen [29] and is thus called LVQ3, formally defined on the facing page.

Algorithm 2. Learning Vector Quantization 3

The input to the algorithm is a labelled codebook $\mathcal{C}$, a set of labelled training vectors T, a window size $\delta \in \langle 0.0, 1.0\rangle$, a fall-off factor $\epsilon \in \langle 0.0, 1.0]$ and the number of training iterations N.

Let $\alpha(t)$ be a monotonically decreasing function with range $\langle 0.0, 1.0]$.

1. $t \leftarrow 0$.
2. While $t < N$:
   (a) Increment counter:
   $$t \leftarrow t + 1.$$
   (b) Choose a random training vector:
   $$\boldsymbol{x} \overset{r}{\leftarrow} T.$$
   (c) Find the two closest codewords:
   $$\boldsymbol{\mu}_i, \boldsymbol{\mu}_j \leftarrow \operatorname*{arg\,min}_{\boldsymbol{\mu}_{i'}, \boldsymbol{\mu}_{i'} \in \mathcal{C}} d(\boldsymbol{\mu}_{i'}, \boldsymbol{x}) + d(\boldsymbol{\mu}_{j'}, \boldsymbol{x}) \quad : \; i' \neq j'.$$
   (d) If neither labels of $\boldsymbol{\mu}_i$ and $\boldsymbol{\mu}_j$ match the label of $\boldsymbol{x}$ then repeat 2.
   (e) Check if $\boldsymbol{x}$ is "in the window":
   $$\min\left(\frac{d(\boldsymbol{\mu}_i, \boldsymbol{x})}{d(\boldsymbol{\mu}_j, \boldsymbol{x})}, \frac{d(\boldsymbol{\mu}_j, \boldsymbol{x})}{d(\boldsymbol{\mu}_i, \boldsymbol{x})}\right) > \delta.$$
   If not, repeat from 2.
   (f) If both labels of $\boldsymbol{\mu}_i$ and $\boldsymbol{\mu}_j$ match the label of $\boldsymbol{x}$ then:
   $$\begin{aligned} \boldsymbol{\mu}_i &\leftarrow \mu_i + \epsilon \cdot \alpha(t) \cdot [\boldsymbol{x} - \boldsymbol{\mu}_i] \\ \boldsymbol{\mu}_j &\leftarrow \mu_j + \epsilon \cdot \alpha(t) \cdot [\boldsymbol{x} - \boldsymbol{\mu}_j]. \end{aligned}$$
   And repeat from 2.
   (g) If necessary swap $\boldsymbol{\mu}_i$ and $\boldsymbol{\mu}_j$ so that the label of $\boldsymbol{x}$ matches that of $\boldsymbol{\mu}_i$.
   (h) Update $\boldsymbol{\mu}_i$ and $\boldsymbol{\mu}_j$:
   $$\begin{aligned} \boldsymbol{\mu}_i &\leftarrow \mu_i + \epsilon \cdot \alpha(t) \cdot [\boldsymbol{x} - \boldsymbol{\mu}_i] \\ \boldsymbol{\mu}_j &\leftarrow \mu_j - \epsilon \cdot \alpha(t) \cdot [\boldsymbol{x} - \boldsymbol{\mu}_j]. \end{aligned}$$
   And repeat from 2.

Chapter 3

Theoretical Analysis of the Side-Channel

This chapter will introduce the theoretical framework for evaluating and comparing physical side-channels introduced by Standaert et al. in [42]. We start out by reciting the assumptions underlying the model behind the framework in section 3.1, followed by one of the major theorems of [42] and a brief discussion of how this may be used to compare side-channels.

3.1 Model Assumptions and Definitions

Firstly, the model in this chapter is based on the following informal axioms, introduced by Micali and Reyzin and here directly quoted from [34]:

1. *Computation, and only computation, leaks information*
   Information may leak whenever bits of data are accessed and computed upon. The leaking information actually depends on the particular operation performed, and, more generally, on the configuration of the currently active part of the computer. However, there is no information leakage in the absence of computation: data can be placed in some form of storage where, when not being accessed and computed upon, it is totally secure.
2. *Same computation leaks different information on different computers*
   Traditionally, we think of algorithms as carrying out computation. However, an algorithm is an abstraction: a set of general instructions, whose physical implementation may vary. In one case, an algorithm may be executed in a physical computer with lead shielding hiding the electromagnetic radiation correlated to the machine's internal state. In another case, the same algorithm may be executed in a computer with a sufficiently powerful inner

battery hiding the power utilised at each step of the computation. As a result, the same elementary operation on 2 bits of data may leak different information: e.g., (for all we know) their XOR in one case and their AND in the other.

3. *Information leakage depends on the chosen measurement*
   While much may be observable at any given time, not all of it can be observed simultaneously (either for theoretical or practical reasons), and some may be only observed in a probabilistic sense (due to quantum effects, noise, etc.). The specific information leaked depends on the actual measurement made. Different measurements can be chosen (adaptively and adversarially) at each step of the computation.
4. *Information leakage is local*
   The information that may be leaked by a physically observable device is the same in any execution with the same input, independent of the computation that takes place before the device is invoked or after it halts. In particular, therefore, *measurable information dissipates*: Though an adversary can choose what information to measure at each step of a computation, information not measured is lost. Information leakage depends on the past computational history only to the extent that the current computational configuration depends on such history.
5. *All leaked information is efficiently computable from the computer's internal configuration*
   Given an algorithm and its physical implementation, the information leakage is a polynomial-time computable function of (1) the algorithm's internal configuration, (2) the chosen measurement, and possibly (3) some randomness (outside anybody's control).

Following these axioms we will now introduce the necessary notation from [34] needed to discuss the framework by Standaert.

Definition 8. An *abstract virtual-memory computer*, or *abstract computer* for short, consists of a collection of special Turing machines, which invoke each other as subroutines and share a special common memory. Each element of an abstract computer is called an *abstract memory Turing machine* (abstract VTM).

$$\mathcal{A} = (A_1, \ldots, A_n)$$

All input and output to the abstract VTM's are arbitrary binary strings. It is important to stress that abstract computers and abstract VTM's are not physical devices, they are only representations of logical computations.

Definition 9. A *physical VTM* is a tuple, $P = (L, A)$ where A is an abstract VTM and L is a *leakage function* as defined below. If $\mathcal{A} = (A_1, A_2, \ldots, A_n)$ is an abstract computer and $P_i = (L_i, A_i)$, then P_i is a physical implementation of A_i and $\mathcal{P} = (P_1, \ldots, P_n)$ is a physical implementation of $\mathcal{A}$.

Example 1. An abstract VTM could be the calculation performed by a cryptographic primitive. Taking RSA as an example, an abstract computer would be the operation

$$c \equiv m^e \pmod{n}$$

whilst the physical VTM would be physical realisations of these operations each having a different leakage function. Example physical VTMs:

1. The Multiply-Square exponentiation algorithm in software.
2. The Montgomery Multiplication algorithm in software.
3. A smart card's signing hardware.

Definition 10. The leakage function $L(C_A, M, R)$ models the leakage from a physical realisation of an abstract VTM. It takes three inputs:

– C_A, the internal configuration of the abstract VTM A.
– M, the setting of the apparatus used to measure the physical observable.
– R, a random string to model the randomness of the measurement.

3.2 Framework

In this section we will introduce the framework of Standaert et al. [42] for analysing the information leakage from physical side-channels. The framework is modelled around recovering well defined *parts* of the key, but, as mentioned in the original article, any secret information that can be modelled by the leakage function can be analysed using the framework.

3.2.1 Attack Model

The attack model is illustrated in figure 3.1 on the next page. The attack is divided into two phases. Informally phase one is where the adversary tries to approximate the probability distribution (pdf) of the leakage function for a part of the key. In phase two, this pdf is used for exploitation.

This paper is concerned about the evaluation of the side-channel itself and not on comparing how efficient different adversaries are at extracting the information. Therefore this section will present the necessary background to compare the leakage between different implementations. [42] also covers comparing the efficiency of different adversaries, but that is outside the scope of this thesis.

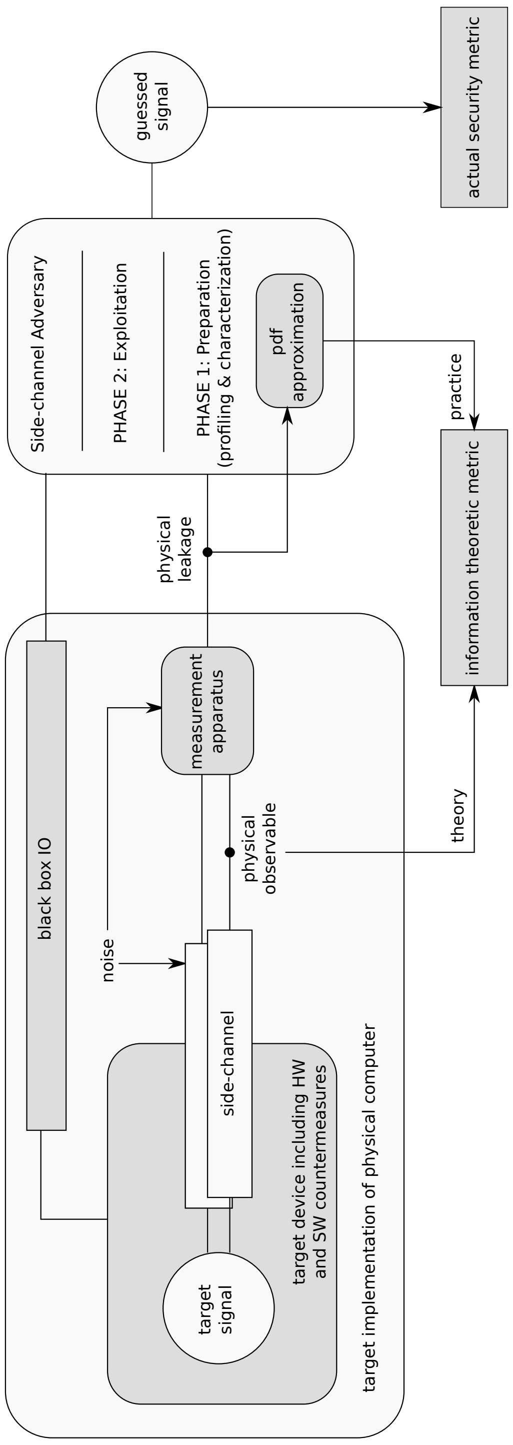


Figure 3.1: Intuitive description of a side-channel key recovery attack ([42] p.6).

3.2.2 Information Theoretic Metrics

As briefly mentioned, the goal of the adversary in phase one of the attack is to approximate the leakage function for a part of the key, we model this partial key information as the *key class* a key belongs to.

Definition 11. Let $k \in \mathcal{K}$ be key from the key set of a cryptographic abstract computer $\mathtt{E_K}$. Let $\gamma : \mathcal{K} \to \mathcal{S}$ be a mapping from the key set to a set of *key classes*, such that $|\mathcal{S}| \ll |\mathcal{K}|$.

An example: $\gamma : \mathcal{K} \to \{0, 1\}$ could partition the key set to keys starting with 0 or 1 respectively.

Next, define a key recovery adversary $\mathtt{A}_{\mathtt{E_K},\mathtt{L}}(\tau, m, q)$ as an algorithm using at most τ time, m memory and q queries to the target computer. The adversary's goal is to guess the correct key class $s = \gamma(k)$, using both the black box information (e.g. cipher-text and/or known plain-texts) and physical information leaked from the side-channel(s) of the physical computer $(\mathtt{E_K}, L)$.

The adversary returns an ordered set $\boldsymbol{g} = (g_1, g_2, \ldots, g_{|\mathcal{S}|})$ which is a permutation of the key classes in $\mathcal{S}$.

Definition 12. The o-th order *success rate* of the adversary is defined as

$$\mathbf{Succ}^{sc-kr-o,\mathcal{S}}_{\mathtt{A}_{\mathtt{E_K},\mathtt{L}}}(\tau, m, q) = \Pr[\mathtt{Exp}^{sc-kr-o}_{\mathtt{A}_{\mathtt{E_K},\mathtt{L}}} = 1] \tag{3.1}$$

Where $\mathtt{Exp}^{sc-kr-o}_{\mathtt{A}_{\mathtt{E_K},\mathtt{L}}}$ is successful if and only if s is in the first o elements of $\boldsymbol{g}$.

```
Experiment $\mathtt{Exp}^{sc-kr-o}_{\mathtt{A}_{\mathtt{E_K},\mathtt{L}}}$ :
    $k \overset{r}{\leftarrow} \mathcal{K}$
    $s = \gamma(k)$
    $\boldsymbol{g} \leftarrow \mathtt{A}_{\mathtt{E_K},\mathtt{L}}$
    if $s \in (g_1, \ldots, g_o)$
        return 1
    else
        return 0
```

For the remainder of this paper we will use order $o = 1$.

Definition 13. The *asymptotic success rate* of an adversary $\mathtt{A}_{\mathtt{E_K},\mathtt{L}}$ is the limit of its success rate as q goes to infinity.

$$\mathbf{Succ}^{sc-kr-o,\mathcal{S}}_{\mathtt{A}_{\mathtt{E_K},\mathtt{L}}}(\tau, m, q \to \infty).$$

3.2.3 Conditional Entropy

This section will introduce Shannon's conditional entropy as a metric to measure the amount of information leaked from a side-channel.

The setup is as follows, let $\boldsymbol{L}_q$ be a stochastic variable of the side-channel leakage observations and let $\boldsymbol{l}_q = (l_1, l_2, \ldots, l_q)$ be a realisation of this variable.

Entropy defined by Shannon [41], is a measure of the information contained in a message, it is used as a tool in for instance data compression. Intuition yields that predictive data have low entropy, whilst completely random data have high entropy.

More formally, Shannon defines the entropy of a discrete stochastic variable X as:

$$H[X] = -\sum_{x \in X} \Pr(x) \cdot \log_2 \Pr(x) \tag{3.2}$$

Now, we are concerned with the entropy for the distribution of $\Pr(S \mid \boldsymbol{L}_q)$ as it would help us decide on the difficulty of determining a key class s given an observation vector $\boldsymbol{l}_q$.

Shannon already defines this conditional entropy.

$$\begin{aligned} H[X \mid Y] &= -\sum_{x \in X} \sum_{y \in Y} \Pr(x, y) \cdot \log_2 \Pr(x \mid y) && (3.3) \\ &= -\sum_{x \in X} \sum_{y \in Y} \Pr(x) \cdot \Pr(y \mid x) \cdot \log_2 \Pr(x \mid y) && \\ &= -\sum_{x \in X} \Pr(x) \sum_{y \in Y} \Pr(y \mid x) \cdot \log_2 \Pr(x \mid y) && (3.4) \\ &= -\sum_{y \in Y} \Pr(y) \sum_{x \in X} \Pr(x \mid y) \cdot \log_2 \Pr(x \mid y) && (3.5) \end{aligned}$$

Now, define the *conditional entropy matrix*

$$\mathbf{H}_S^q = \left(h_{s,s^*}^q\right)_{s,s^* \in S} \tag{3.6}$$

where

$$h_{s,s^*}^q = -\sum_{\boldsymbol{l}_q} \Pr(\boldsymbol{l}_q \mid s) \cdot \log_2 \Pr(s^* \mid \boldsymbol{l}_q). \tag{3.7}$$

S is a discrete stochastic variable of the previously targeted key class $\mathcal{S}$.

Also, note that:

$$H[S \mid \boldsymbol{L}_q] = -\sum_{S} \Pr(s) \sum_{\boldsymbol{l}_q} \Pr(\boldsymbol{l}_q \mid s) \cdot \log_2 \Pr(s \mid \boldsymbol{l}_q) = \mathop{\mathbf{E}}_{s}(h_{s,s}) \tag{3.8}$$

is exactly Shannon's conditional entropy from (3.4).

Finally, we define the *mutual information* as,

$$I(S, \boldsymbol{L}_q) = H[S] - H[S \mid \boldsymbol{L}_q]. \tag{3.9}$$

Mutual information is a measure of how much of the information in S is reflected in $\boldsymbol{L}_q$.

It is important to note that in general, we do not know the probability distribution of $\Pr(L_q \mid S)$, but rather we estimate the probability distribution $\hat{\Pr}(\tilde{L}_q \mid S)$ using a practical number of samples.

3.3 Measuring the Information Leakage

Before we recite one of the theorems of [42] that will be the climax of this chapter, we must first define a few more concepts.

Definition 14. A first-order *Bayesian side-channel adversary* (choosing only one key class from $\mathcal{S}$) is an adversary that always selects $\arg\max_{s^*} \Pr(s^* \mid \boldsymbol{l}_q)$.

Definition 15. An approximated leakage distribution $\Pr(\tilde{\boldsymbol{L}}_q \mid S)$ is considered *sound* if the asymptotic success rate of a first-order Bayesian side-channel adversary is one.

Theorem 1. Assuming independent leakages for the different queries in a side-channel attack, an approximated leakage probability distribution $\hat{\Pr}(\tilde{\boldsymbol{L}}_q \mid S)$ is sound if and only if the conditional entropy matrix evaluated in an unbounded exploitation phase is such that the diagonal element is the smallest in every row of the approximated entropy matrix $\hat{\mathbf{H}}^q_{\mathcal{S}}$. That is,

$$\arg\min_{s^*} \hat{h}^q_{s,s^*} = s, \forall s \in \mathcal{S}. \tag{3.10}$$

Proof. Let s be the target key class, and $(\boldsymbol{l}_q)_p = (\boldsymbol{l}_{q,1}, \boldsymbol{l}_{q,2}, \ldots, \boldsymbol{l}_{q,p})$ be p realisations of a q-queries leakage vector $\boldsymbol{L}_q$.

Now, a Bayesian adversary using targeting s using the leakage data $(\boldsymbol{l}_q)_p$ will be successful if and only if

$$s = \arg\max_{s^*} \hat{\Pr}(s^* \mid (\tilde{\boldsymbol{l}}_q)_p)$$
$$s = \arg\max_{s^*} \frac{\hat{\Pr}((\tilde{\boldsymbol{l}}_q)_p) \mid s^*) \cdot \Pr(s^*)}{\hat{\Pr}((\tilde{\boldsymbol{l}}_q)_p)}.$$

Assume that $\Pr(s^*)$ is uniformly distributed. Note that since $\hat{\Pr}((\tilde{\boldsymbol{l}_q})_p))$ only depends on the correct key class s it is independent of s^*. This yields,

$$s = \arg\max_{s^*} \hat{\Pr}((\tilde{\boldsymbol{l}_q})_p) \mid s^*).$$

Since we assume that each measurement in $(\tilde{\boldsymbol{l}_q})_p$ is independent, we may rewrite the above as:

$$s = \arg\max_{s^*} \prod_{i=1}^{p} \hat{\Pr}(\tilde{\boldsymbol{l}}_{q,i} \mid s^*).$$

Now, as we are looking at an unbounded exploitation phase (asymptotic attack), the number of leakage vectors p is not bounded and the measurement vector $(\tilde{\boldsymbol{l}_q})_p$ is in fact a trace from the real probability distribution $\Pr(\boldsymbol{L}_q, \mid s)$. Therefore as $p \to \infty$ each unique leakage vector in $\boldsymbol{l}_q \in (\tilde{\boldsymbol{l}_q})_p$ will repeat $p \cdot \Pr(\boldsymbol{l}_q \mid s)$ times. Thus, an asymptotic attack is successful if and only if:

$$\begin{aligned}
s &= \arg\max_{s^*} \prod_{\boldsymbol{l}_q} \hat{\Pr}(\boldsymbol{l}_q \mid s^*)^{p \cdot \Pr(\boldsymbol{l}_q \mid s)} \\
s &= \arg\max_{s^*} \prod_{\boldsymbol{l}_q} \left(\hat{\Pr}(\boldsymbol{l}_q \mid s^*)^{\Pr(\boldsymbol{l}_q \mid s)}\right)^p \\
s &= \arg\max_{s^*} \prod_{\boldsymbol{l}_q} \hat{\Pr}(\boldsymbol{l}_q \mid s^*)^{\Pr(\boldsymbol{l}_q \mid s)} \\
s &= \arg\max_{s^*} \prod_{\boldsymbol{l}_q} \hat{\Pr}(s^* \mid \boldsymbol{l}_q)^{\Pr(\boldsymbol{l}_q \mid s)} \\
s &= \arg\max_{s^*} \sum_{\boldsymbol{l}_q} \Pr(\boldsymbol{l}_q \mid s) \cdot \log_2 \hat{\Pr}(s^* \mid \boldsymbol{l}_q) \\
s &= \arg\min_{s^*} -\sum_{\boldsymbol{l}_q} \Pr(\boldsymbol{l}_q \mid s) \cdot \log_2 \hat{\Pr}(s^* \mid \boldsymbol{l}_q). \qquad (3.11)
\end{aligned}$$

Now, we can observe that the sum (3.11) equals that of (3.7) save for the approximated probability in the logarithmic factor. Thus, it exactly matches how the conditional entropy is estimated in practice. Therefore, if the previous condition holds for all key classes s, the Bayesian side-channel is asymptotically successful and vice versa. □

Remark In practice, $\Pr(s \mid \boldsymbol{l}_q)$ is estimated as:

$$\hat{\Pr}(s \mid \boldsymbol{l}_q) = \frac{\hat{\Pr}(\boldsymbol{l}_q \mid s) \cdot \Pr(s)}{\sum_{s^*} \hat{\Pr}(\boldsymbol{l}_q \mid s^*) \cdot \Pr(s^*)}. \qquad (3.12)$$

3.4 Consequences of Theorem 1

It turns out that under certain conditions[1] theorem 1 suffices to discuss the security of a side-channel. More formally the conditions are when $|\mathcal{S}| = 2$. and the leakage function $L(C_A, M, R) = L'(C_A, M) + L''(R)$ where $L''(R) \sim N(0, \sigma)$. That is the randomness in the leakage function is Gaussian with mean 0 and standard deviation σ.

This gives rise to the following two lemmas [42].

Lemma 1. In a key recovery side-channel attack exploiting a univariate Gaussian leakage distribution of a single query. The residual entropy of a Bayesian attack choosing between exactly two target key classes is a monotonously decreasing function of the single query (hence multi query) success rate against s.

Proof. Let us consider univariate Gaussian leakage and let $|\mathcal{S}| = 2$. Now, without loss of generality, assume the distribution of the correct key class s to have mean 0 and that of the wrong key class to have mean δ. The standard deviation in either case is σ.

Now, assuming a Bayesian adversary, the success rate for the attack will be

$$\mathbf{Succ}^{sc-kr-1,s}_{\mathtt{A}_{\mathtt{E_K}},\mathtt{L}}(\delta, \sigma) = \int_{-\infty}^{\delta/2} \mathbf{N_x}(0, \sigma) dx$$

where

$$\mathbf{N_x}(\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right).$$

The corresponding residual entropy equals

$$h_{s,s}(\delta, \sigma) = -\int_{-\infty}^{\infty} \mathbf{N_x}(0, \sigma) \cdot \log_2 \frac{\mathbf{N_x}(0, \sigma)}{\mathbf{N_x}(0, \sigma) + \mathbf{N_x}(\delta, \sigma)}\, dx$$

Define a change of variables first $u = x/\sigma$ then $z = \delta/\sigma$:

$$\mathbf{Succ}^{sc-kr-1,s}_{\mathtt{A}_{\mathtt{E_K}},\mathtt{L}}(\delta, \sigma) = \int_{-\infty}^{\delta/2\sigma} \mathbf{N_u}(0, 1)\, du$$

$$h_{s,s}(\delta, \sigma) = -\int_{-\infty}^{\infty} \mathbf{N_u}(0, 1) \cdot \log_2 \frac{\mathbf{N_u}(0, 1)}{\mathbf{N_u}(0, 1) + \mathbf{N_u}(\delta/\sigma, 1)}\, du$$

$$\mathbf{Succ}^{sc-kr-1,s}_{\mathtt{A}_{\mathtt{E_K}},\mathtt{L}}(z) = \int_{-\infty}^{z/2} \mathbf{N_u}(0, 1)\, du$$

$$h_{s,s}(z) = -\int_{-\infty}^{\infty} \mathbf{N_u}(0, 1) \cdot \log_2 \frac{\mathbf{N_u}(0, 1)}{\mathbf{N_u}(0, 1) + \mathbf{N_u}(z, 1)}\, du.$$

[1]Standaert et al. also considers more general assumptions, but this is unnecessary for our discussion.

Finally, observe that the success rate is a monotonously increasing function of z and that the residual entropy is a decreasing function of z, which concludes the proof. □

Furthermore, in a multivariate leakage distribution the following lemma holds if each leakage component shares the same standard deviation.

Lemma 2. *In a key recovery side-channel attack exploiting a multivariate Gaussian leakage distribution with independent leakage samples having the same standard deviation. The residual entropy of a Bayesian attack choosing between exactly two target key classes is a monotonously decreasing function of the single query (hence multi query) success rate against s.*

The proof of lemma 2 is similar to that of lemma 1 and is outlined in [42].

One can therefore conclude that in the case of Gaussian noise and a key class size of 2, the higher the conditional entropy $H[S \mid \boldsymbol{L}_q]$ the lower the success rate of a Bayesian adversary will be.

Chapter 4

Proof of Concept

In this section we aim to answer the following questions:

1. Is it possible to use the instruction cache as a side-channel on an ARM platform?
2. What (if any) are the added challenges of attacking an ARM platform as compared to previous work on x86?

In order to answer both questions, a prototype experiment will be performed. In short this experiment attempts to solve the problem:

```
if ( bit == 1 ) {
    foo();
}
```

Observing an execution of the code above, determine whether the function `foo()` was called. This should be determined by observing the instruction cache exclusively.

4.1 Summary of the Target Platform

The platform used for all experiments in this paper is presented in the table below.

Platform	
Hardware:	Hardkernel Odroid X
Hardware Revision:	0.4 20120808
Operating System:	ARM ARCH Linux
Kernel Version:	3.8.13.21-2-ARCH #1 SMP PREEMPT
Kernel Build Date:	Thu Apr 24 19:25:30 MDT 2014

Details	
Chipset:	Samsung Exynos 4412
Processor:	4x Cortex-A9 Quad Core 1.4Ghz
L1 (Data):	32 KB per core
L1 (Instruction):	32 KB per core
L2:	1 MB
Main Memory:	1005 MB
Cache Line Size:	32 bytes

L1 Instruction Cache Details	
Associativity	4
Cache Sets	256
Replacement Policy:	Random Round Robin

Table 4.1: Important platform parameters.

Note on Random Round Robin

The fact that the L1-I cache uses a Random Round Robin[1] (RRR) replacement policy introduces extra noise into the measurements. As can be seen in figure 4.1 this causes cache-misses to bleed into subsequent observations in a seemingly random manner. In a Least Recently Used or First In First Out replacement pattern, we would expect the cache to be completely filled with the spy's instructions after one iteration, which is not the case for RRR.

What is actually happening is that the spy, when trying to access an instruction which is not currently in the cache, will evict a random cache line from the relevant cache set. This cache line may very well contain another part of the spy's

[1]Random Round Robin replacement will evict a pseudo randomly chosen cache line from a cache set.

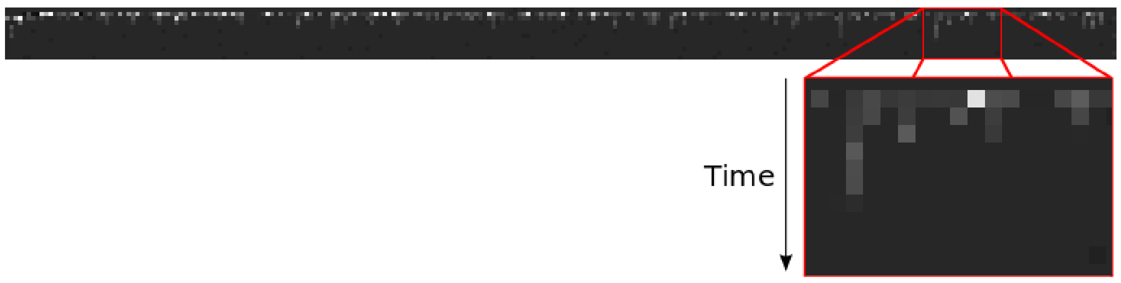


Figure 4.1: The figure show part of a raw timing measurement on the target platform. The bleeding of cache misses into subsequent measurements is clearly visible in the excerpt. More white denote longer access time.

instructions. Thus, we could get two or even more cache misses as a result of one alien cache line in the cache set, or even subsequent measurements as shown in figure 4.1.

Luckily, the use of RRR replacement is of minor consequence for the success of the attack as its effect on the measurement is easily removed by letting the spy run for more rounds between each measurement, thus eventually evicting all alien cache lines.

4.1.1 Timing on ARM

One of the first challenges in implementing the instruction cache side-channel on ARM is getting access to accurate high-precision timing. How to access these timing registers differ between different versions of ARM CPUs. This paper will limit itself to discussing timing on a Cortex A9. Similar, if not identical, procedures can be followed on both older and newer cores.

On x86 the core local cycle count is easily readable through the `RDTSC`-instruction. On ARMv7 there is no specialised instruction to retrieve the cycle count. Instead we have to use the more general `MRC`-instruction to read from the performance cycle count register (`CCNT`). Unfortunately, access to this register is disabled by default at the lowest privilege level. It is, however, possible to enable access to this register, even to user space, from a higher privilege level.

The timing data in this paper are all measured using the `MRC`-instruction from user-space after enabling access to it from a loadable kernel module. Source code for enabling user-space timing is available in appendix A.

It is possible that some platforms already expose or will expose the `CCNT` register in the future as all that is needed to expose it is to run a small number of instructions from a higher privilege level during boot or from a driver/module running in privileged mode.

Please note that restricted access to the performance timing register is not sufficient in itself to close the instruction cache side-channel. It may be possible for an attacker to get sufficiently accurate timing from other sources.

4.2 Method

For simplicity, the experiment in this section will simulate ideal conditions where the spy is always run immediately after the function in question. Also, in order to clear the noise introduced by the RRR replacement policy, the spy is run for 25 iterations prior to the call to `foo` [2].

The timing data was sampled using a loop like the one shown below, where `input` is an arbitrary bit string.

```
for (int bit : input ) {
   run_spy( 25 ); // Populate the cache, not measured
   if ( bit == 1 ) {
      foo();
   }
   run_spy( 1 );  // measured.
}
```

Throughout this section, we are considering a 2-target attack, i.e. $|\mathcal{S}| = 2$. We are at any given time only considering one bit of information. The measured output from the leakage function $\tilde{\boldsymbol{L}}_{256}$ is a 256 dimensional vector of timing data. Each component measures a predetermined cache set as a positive integer, with all values above 255 clamped to 255. Parts of the data in $\tilde{\boldsymbol{L}}_{256}$ will be discarded as explained below.

Now, the attack was executed as follows:

Preparation step

1. Generate training data by running the spy on known input (10 separate instances measuring 10000 iterations for each $s \in \{0, 1\}$).
2. Cull dead cache sets, see section 4.2.1.
3. Create and train a SOM using the reduced training data.
4. Tune the resulting VQ using the LVQ3 algorithm.

Exploitation step

1. Run the spy on "unknown" input.

[2] In this experiment the function `foo` is actually `cos(·)` from the standard C-library.

2. Reduce dimensions to the previously determined cache sets.
3. Encode the input using the previously trained VQ.

Finally, the result is compared to the "unknown" input.

4.2.1 Culling Dead Data

As we are measuring every cache set, we expect to see a lot of measurements that do not carry any information about the correct key class. Rather, we expect these dead cache sets to slow down and even hamper our analysis. Below, we present two methods to identify the live cache sets prior to building the vector quantizer.

Timing Mean and the Central Limit Theorem

As can be seen in figure 4.2 on page 36, the mean timing value of the cache sets show a clear tendency of being a good metric to identify information bearing cache sets. However, as the variance of the timings can be quite large, relying purely on the mean could result in a too conservative culling.

Luckily, the problem could be taken directly from undergraduate textbook. By the central limit theorem: The estimated mean will follow a Gaussian distribution, $\bar{\boldsymbol{x}} \sim N(\mu, \frac{\sigma}{\sqrt{n}})$, where n is the number of measurements. We can therefore use a simple two-sided test to check if the means are equal. If the difference is significant, we use that cache set, otherwise we discard it.

In the practical results below, we use this metric to cull dead data.

Mutual Information

Mutual information is another metric that can be used for this purpose. Simply by choosing the cache sets with the highest mutual information as the live. Note, however, that choosing too few cache sets could potentially hamper the analysis in cases where the target process is at an *unusual alignment*, explained in the next section.

In our experiments, we tried using only the cache sets carrying mutual information larger than 0.1. Doing so, however, yielded poorer results than using culling according to the central limit theorem. Thus, great care has to be made when using this as a culling mechanism as even a little information may help the final result.

4.3 Theoretical Analysis

In this section we aim to analyse the quality of the information in the side-channel proposed above. This is separate from the actual implementation of the exploitation step: Here we aim to formally evaluate the side-channel according to the framework presented in chapter 3, whilst the success rate and details of the exploitation step is presented in section 4.4 on page 42. Thus this section answers the question "Can we exploit the information leaked from the side-channel?"

Initial Analysis

Firstly, let us consider the mean and variance for the timing of each cache set, presented in figures 4.2 and 4.3.

For small functions we do not expect much of the cache to be polluted, thus culling cache sets that do not carry information should greatly ease the later analysis. Another noteworthy point is that the mean values seen in figure 4.2 are from several independent executions of the crypto-process. Further inspection into the raw data reveals that the cache set footprint seems to vary between different executions of the program, this is due to a technique called address space layout randomisation. In short the OS will randomise where in memory it loads the executable. This is done to increase the difficulty of certain buffer overflow exploits. The rearranged data will, however, still retain alignment to memory page boundaries, which is sufficient for our analysis[3].

Furthermore, as can be seen in figure 4.3, there is clearly a lot of noise in the measurements. Additionally, the data set where are known cache misses seems to have a slightly higher variance than the sets without a forced cache miss.

The source of this variance becomes more clear once we take a look at the distribution of the measurements in a single cache set. Below, cache set 148 was chosen as a candidate seeming to carry significant mutual information. A histogram of the data is shown in figure 4.4. There appear to be 3–4 distinct peaks where most access time measurements reside. This is most likely a product of one or more cache misses in the measurement for this cache set, or even instances where the memory system has had to access lower level memory than L2. Recall that the L1 instruction cache in this experiment has random round robin replacement, so we expect some randomness in the number of cache misses.

Another, perhaps even more interesting point is that not all measurements of $s = 0$ seem to result in a cache hit. This could be a result of various effects, such as OS-ticks, hardware interrupts, preemptive scheduling of other processes etc.

[3]The cache set-alignment boundary on this platform is $2 \cdot PageSize$, we will therefore have only two separate footprints.

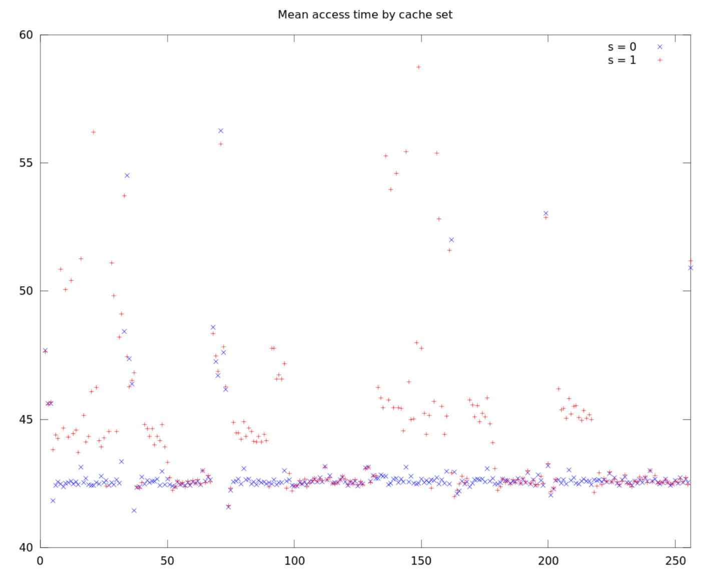


Figure 4.2: Plot shows mean value for spy measurements per cache set. Cache sets with frequent cache misses are clearly visible.

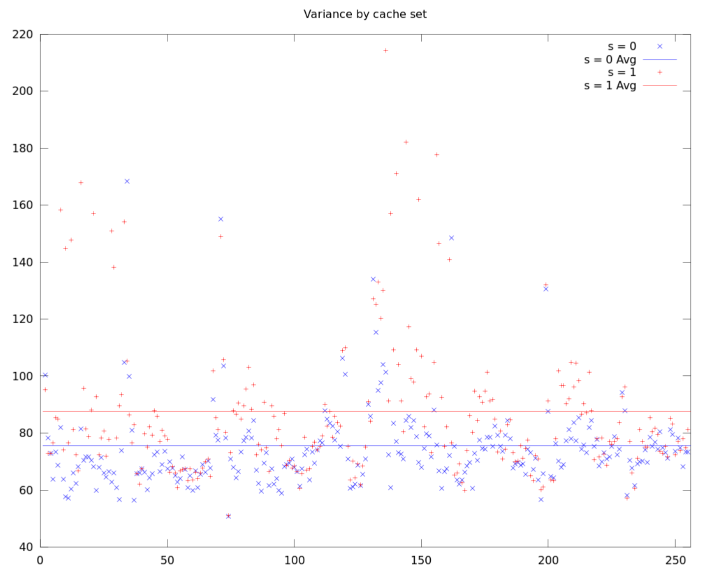


Figure 4.3: Plot shows the variance of spy measurements per cache set.

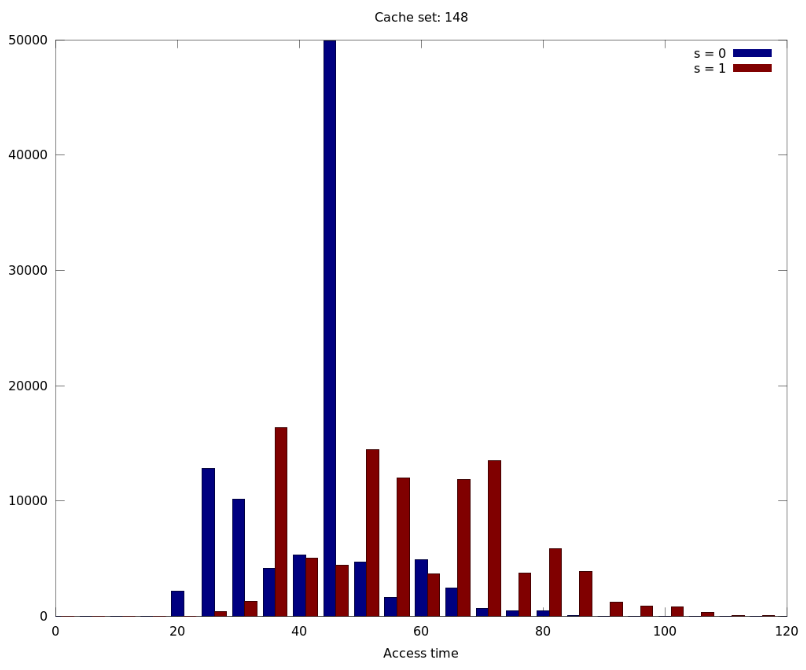


Figure 4.4: Histogram of cache access measurements for cache set 148.

Estimating $\Pr(\boldsymbol{L}_q \mid S)$

For simplicity, we will only consider the univariate distribution of the measurement in cache set 148. The results for other cache sets are presented in table 4.2 on page 40 and we cannot assume that they are independent, but separate analysis should still yield a lot of information of the side-channel.

After measuring the 100000 samples for each $s \in S$, the estimated probability distribution $\hat{\Pr}(\tilde{\boldsymbol{L}}_q \mid S)$ is easily estimated using a frequency table. The result is presented in figure 4.5. The access time axis is cropped at 120 as the the probability of access times longer than that tend towards zero.

Furthermore, the distribution $\hat{\Pr}(\tilde{\boldsymbol{L}}_q \mid S)$ is used to calculate $\hat{\Pr}(S \mid \tilde{\boldsymbol{L}}_q)$ using the formula (3.12) on page 27. The results are shown in figure 4.6. For measurements between 20–120 there is a trend that may be used to determine s. As expected, measurements outside the range of commonly observed values yield unpredictable results.

Entropy

Shannon's conditional entropy for the measurements of cache set 148 is presented below.

First, the conditional entropy matrix is calculated from (3.6):

$$\hat{\mathbf{H}}^1_{S,148} = \begin{bmatrix} 0.44568 & 3.33847 \\ 3.41892 & 0.44405 \end{bmatrix}.$$

Thus, from theorem 1 on page 26 it is clear that the estimated probability distribution of $\hat{\Pr}(\boldsymbol{L}_q \mid S)$ for cache set 148 is sound.

Unfortunately, as the noise in our measurements is not Gaussian, the lemmas of chapter 3 do not apply. However, we may still calculate the mutual information of S that we have captured by our estimation of $\Pr(\boldsymbol{L}_q \mid S)$.

$$H[S \mid \tilde{\boldsymbol{L}}_q] = \underset{s}{\mathbf{E}}(\hat{h}_{s,s}) = 0.44487.$$
$$H[S] = 1$$

Which yields the mutual information:

$$I(S \mid \tilde{\boldsymbol{L}}_q) = 0.55513.$$

The results for the remaining cache sets are shown in table 4.2 on page 40. In this experiment all but one of the cache sets have sound approximations for $\tilde{\boldsymbol{L}}_q$, but cache sets having mutual information less than 0.01 have been omitted for compactness.

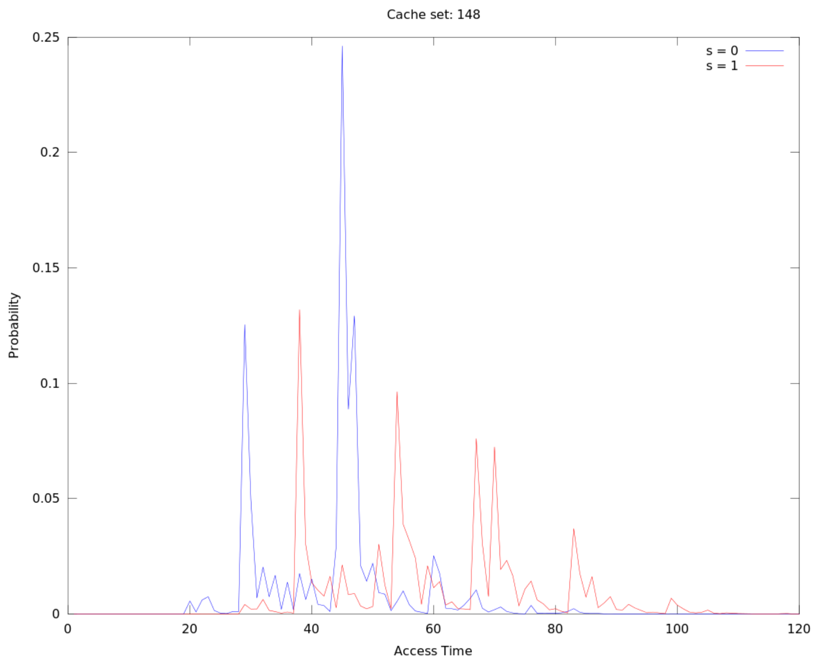


Figure 4.5: Estimated probability distribution $\hat{\Pr}(\tilde{\boldsymbol{L}}_q \mid S)$.

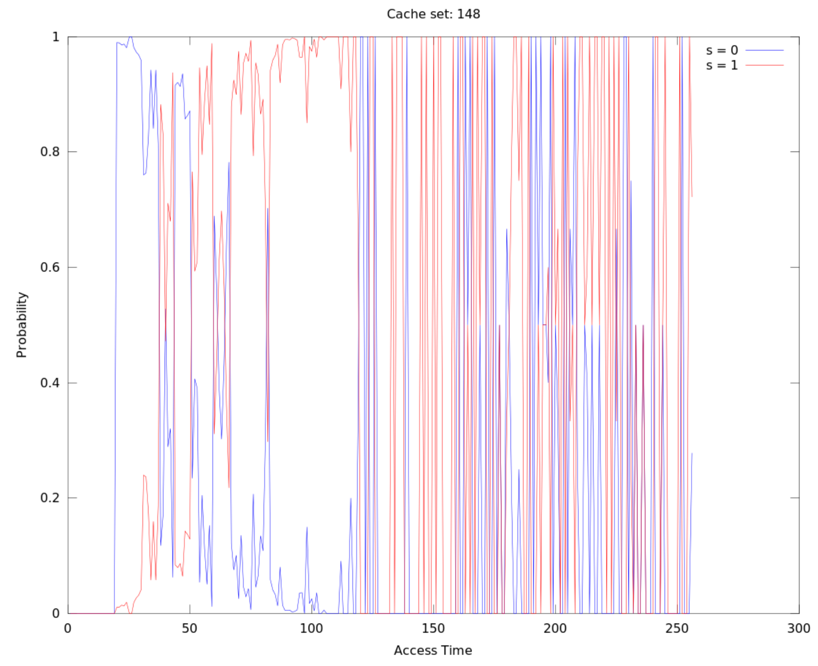


Figure 4.6: Estimated probability distribution $\hat{\Pr}(S \mid \tilde{\boldsymbol{L}}_q)$.

Cache set	$I(S, \tilde{\boldsymbol{L}}_q)$	Cache set	$I(S, \tilde{\boldsymbol{L}}_q)$	Cache set	$I(S, \tilde{\boldsymbol{L}}_q)$
0	Not Sound	78	0.0120	153	0.0113
4	0.0125	82	0.0139	154	0.0265
5	0.0131	84	0.0116	155	0.3425
6	0.0129	85	0.0115	156	0.3483
7	0.1745	86	0.0103	157	0.0187
8	0.0183	87	0.0148	158	0.0153
9	0.1673	88	0.0172	159	0.0120
11	0.1685	89	0.0245	160	0.3231
12	0.0104	90	0.0703	161	0.3116
13	0.0156	91	0.0762	168	0.0281
15	0.1580	92	0.0615	169	0.0271
16	0.0156	93	0.0522	170	0.0240
18	0.0158	94	0.0515	171	0.0280
19	0.0335	95	0.0369	172	0.0229
20	0.4681	132	0.0290	173	0.0218
21	0.0294	133	0.0263	174	0.0232
22	0.0137	134	0.0256	175	0.0169
26	0.0110	135	0.3335	176	0.0128
27	0.1822	136	0.0317	203	0.0279
28	0.1826	137	0.3259	204	0.0269
29	0.0183	138	0.0195	205	0.0235
30	0.0719	139	0.3305	206	0.0249
31	0.0609	140	0.0199	207	0.0186
32	0.1190	141	0.0260	208	0.0169
33	0.1324	142	0.0222	209	0.0176
36	0.0521	143	0.3289	210	0.0251
40	0.0121	144	0.0304	211	0.0197
41	0.0118	145	0.0160	212	0.0227
42	0.0111	146	0.0274	213	0.0239
43	0.0160	147	0.0685	214	0.0217
45	0.0109	148	0.5551	215	0.0173
46	0.0116	149	0.0456	216	0.0175
75	0.0135	150	0.0289		
76	0.0128	151	0.0191		
77	0.0114	152	0.0216		

Table 4.2: Cache sets with mutual information $I(S, \tilde{\boldsymbol{L}}_q) > 0.01$.

Remarks on Cache Set 0 As can be seen in table 4.2 cache set 0 seems to be a special case as the only cache set that is not sound. This is because we store the measurements in unsigned bytes and clamp all data exceeding this to 255. For cache set 0 *all data* seem to violate this constraint, therefore, cache set 0 has mean 255 and variance 0, regardless of the value of s. We have been unable to isolate the cause of this, but it is of no practical importance as cache set 0 is automatically culled.

Conclusion

After looking at the data in table 4.2 we conclude that there is exploitable information in in the side-channel. As expected, there are certain cache sets that carry significantly more information about S than others and are the primary contributors to the leakage in the side-channel. Also, as will be shown in the next section, this leakage is more than enough to create an adversary with significant advantage.

4.4 Practical Results

As is seen in table 4.3 the success rate of this experiment was over 97%.

Input parameters	
SOM Dimensions:	25
Training samples:	10 x 10000 measurements per key class
Challenge samples:	10000

Results	
Classification Rate on Training Data:	0.973
Classification Rate on Challenge Data	0.972

Table 4.3: Results of experiment.

From this we conclude that the first order success rate of this adversary is:

$$\mathbf{Succ}^{sc-kr-1,s}_{\mathtt{A}_{\mathtt{E}_{\{0,1\}}},\mathtt{L}} = 0.972. \tag{4.1}$$

Thus we have created an adversary with advantage:

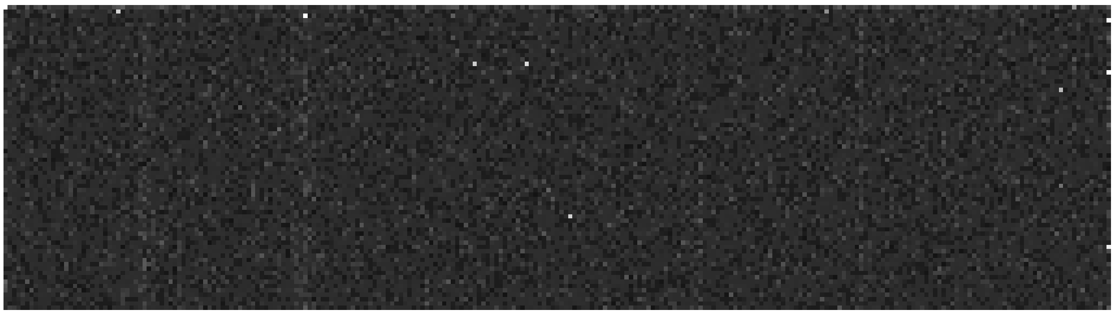
$$Adv(\mathtt{A}_{\mathtt{E}_{\{0,1\}}}) = |P(\mathtt{A}_{\mathtt{E}_{\{0,1\}}} \to 1 \mid s = 1) - 1/2| = 0.472. \tag{4.2}$$

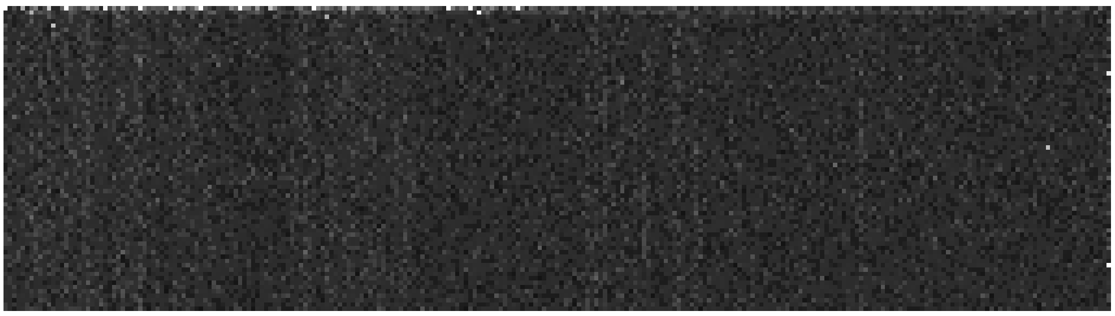
Since both SOM and LVQ are randomised algorithms the exact hit-rates vary between subsequent runs.

For completeness, visualisations of an extract of the raw timing data are shown in figure 4.7 on the facing page.

4.5 Conclusion

It is clear from the experiment above that we are able to determine whether a function was called with a significant advantage in this ideal context. The theoretical analysis shows that there is exploitable information in the signal exposed by the instruction cache, and the practical attack succeeds with a good advantage. Next up is trying the attack on an implementation of an actual crypto-system.


(a) Function not called.


(b) Function called.

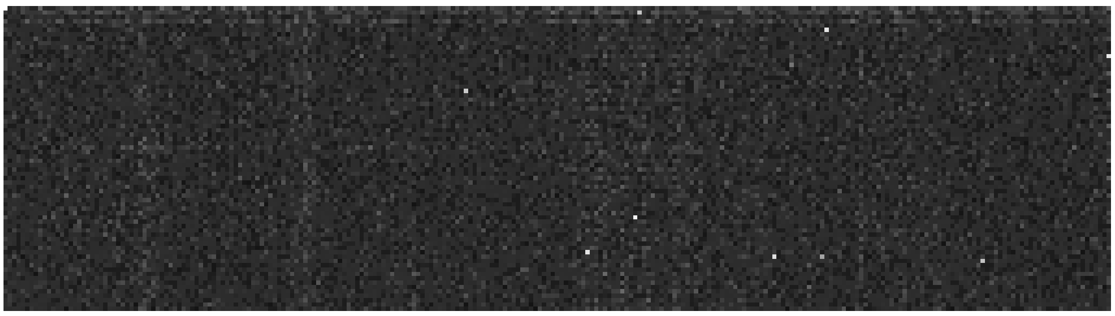

(c) Alternating pattern.

Figure 4.7: Visualisations of the raw timing data. Cache sets are shown along the horizontal axis, and subsequent runs along the vertical axis. The "keys" are "000000...", "111111..." and "010101..." for (a), (b) and (c) respectively. In (b) and (c) it is possible to see more vertical lines in the than in (a), this is the pattern we are searching for. In (c) the striping pattern from the alternating key is clearly visible even to the naked eye. (Depending on the quality of the printout this may not be clearly visible, if that is the case please consult the electronic copy of the thesis.).

Chapter 5

Synchronous Attack on RSA

In this chapter we perform the next logical step and employ the same procedure as in the previous chapter, only against an implementation of an actual crypto-system. We will target a trivial implementation of RSA, using the multiply-square algorithm, which is unfortunately still encountered in the wild, despite being vulnerable to a plethora of known attacks.

The work presented in this chapter is essentially combining parts of [1], [2] and [8] to implement an effective attack.

5.1 Method

As mentioned, we will be attacking an implementation of the multiply-square algorithm. This algorithm, although repeatedly shown to be vulnerable against an array of side-channel attacks is still in use in common crypto-software. GnuPG [48] and some configurations of OpenSSL [2] still use this technique. In this experiment we have implemented the algorithm ourselves using OpenSSL's BIGNUM library. Key generation and other facilities are taken from the OpenSSL library.

The phases of the attack are very similar to the proof of concept in the previous section:

Preparation step

1. Generate 100 random, known RSA-keys.
2. Observe and measure 1 decryption of a random message for each key.
3. Cull dead cache sets, see section 4.2.1.
4. Create and train a SOM using the reduced training data.
5. Tune the resulting VQ using the LVQ3 algorithm.

Exploitation step

1. Generate 1 random challenge RSA-key.
2. Observe and measure 1 decryption of a random message for the challenge key.
3. Reduce dimensions to the previously determined cache sets.
4. Encode the measurements using the previously trained VQ.

The spy function is identical to that in the previous chapter and is listed in appendix B. The full source code for the decryption algorithm is available in appendix C and shows how we call the spy from within the crypto-process.

5.2 Results and Discussion

In this section we present the formal analysis followed by the practical results.

5.2.1 Formal Analysis

As before, we used the mean access time as the metric to exclude cache sets from the training data. This operation identified 239 out of the 256 cache sets as information carriers. We also note that the variance of all cache sets is much higher in this second experiment as compared to the previous chapter. This is not unexpected as the functions we are measuring are much larger and run significantly longer than the simple experiment in the previous chapter. The mean and variance of each measurement is presented in figures 5.1 and 5.2 on the next page.

For the formal analysis, we again follow the framework presented in chapter 3. As before we are looking at a two-target attack, $S = \{0, 1\}$. The mutual information is plotted in table 5.1 on page 48. Furthermore, the estimated probability distributions $\hat{\Pr}(\tilde{\boldsymbol{L}}_q \mid S)$ and $\hat{\Pr}(S \mid \tilde{\boldsymbol{L}}_q)$ for cache set 241, which has the highest mutual information, are shown in figures 5.3 and 5.4 on page 47.

From the data presented thus far there are a few interesting observations: When comparing the data from this chapter to that of chapter 4 we notice that the variance is higher, there are fewer cache sets with high mutual information and there are more cache sets that appear to carry some information.

Despite this, the information is sufficient to create an effective adversary as will be shown in the next section.

5.2.2 Practical Results

As seen in table 5.2 the success rate of this experiment was over 94%. And we conclude that the first order success rate of this adversary is:

$$\mathbf{Succ}^{sc-kr-1,s}_{\mathtt{A}_{\mathtt{E}_{\{0,1\}}},\mathtt{L}} = 0.944. \tag{5.1}$$

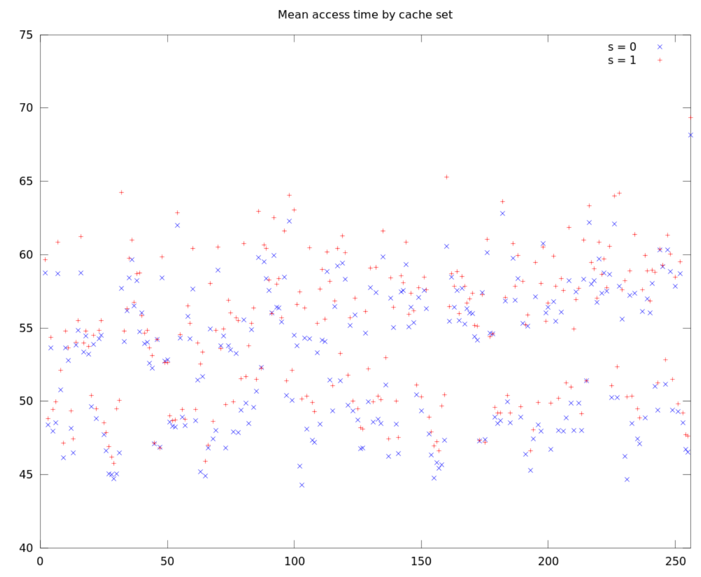


Figure 5.1: Plot shows mean value for spy measurements per cache set. Differences in the mean access time of individual cache sets are visible, but there is no clear trend as in figure 4.2.

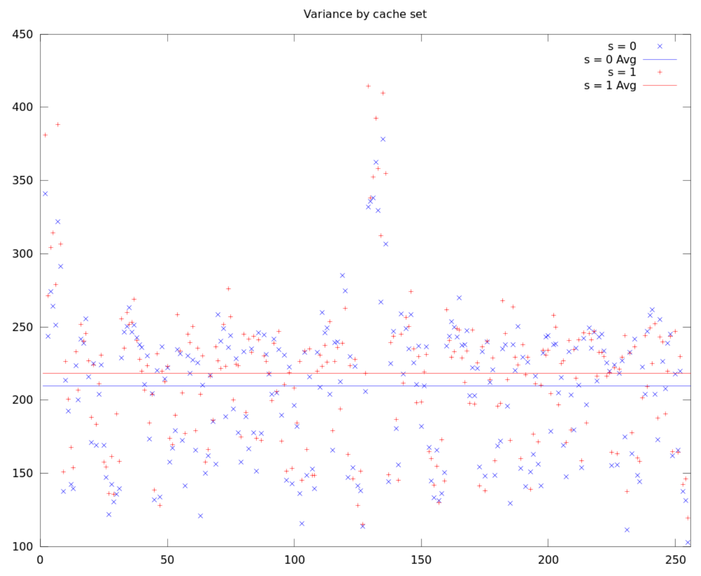


Figure 5.2: Plot shows the variance of spy measurements per cache set.

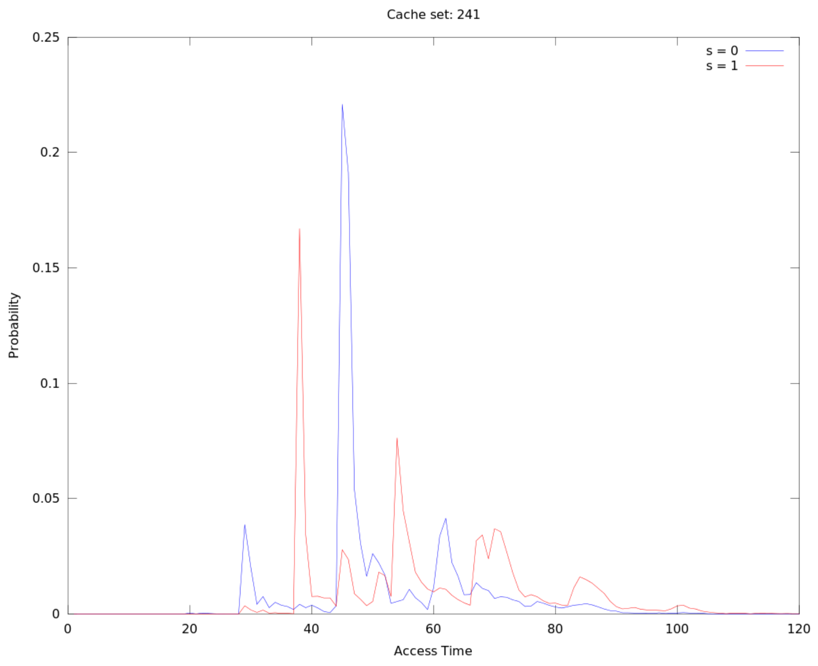


Figure 5.3: Estimated probability distribution $\hat{\Pr}(\tilde{\boldsymbol{L}}_q \mid S)$.

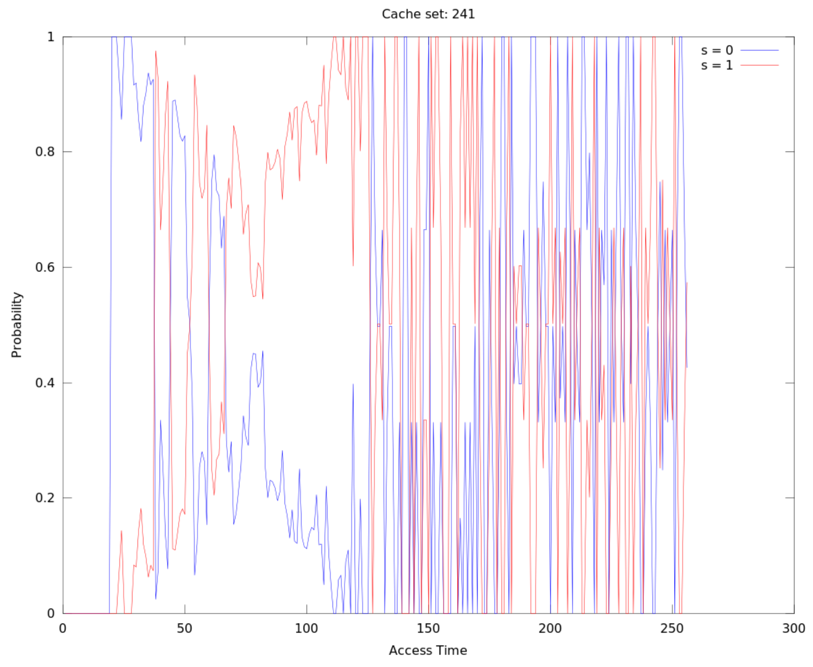


Figure 5.4: Estimated probability distribution $\hat{\Pr}(S \mid \tilde{\boldsymbol{L}}_q)$.

Cache set	$I(S, \tilde{\boldsymbol{L}}_q)$	Cache set	$I(S, \tilde{\boldsymbol{L}}_q)$	Cache set	$I(S, \tilde{\boldsymbol{L}}_q)$
0	Not Sound	95	0.0443	197	0.0368
15	0.0308	97	0.0110	200	0.0199
26	0.0119	98	0.0178	201	0.0150
27	0.0114	99	0.1232	203	0.0125
28	0.0162	100	0.0482	204	0.0162
29	0.0382	101	0.3144	205	0.2655
30	0.0334	102	0.0714	206	0.0222
31	0.0753	103	0.0205	207	0.0246
34	0.0137	104	0.0167	208	0.0104
35	0.0141	105	0.0522	209	0.1039
38	0.0883	106	0.0175	211	0.2517
47	0.0166	107	0.0248	212	0.0227
59	0.0340	108	0.0195	213	0.0269
61	0.0128	109	0.1562	216	0.0105
62	0.1606	110	0.0600	219	0.0162
66	0.0397	112	0.0251	223	0.0209
68	0.1432	113	0.2757	225	0.0115
69	0.0427	114	0.0309	226	0.0184
72	0.0191	118	0.0175	227	0.0818
73	0.0154	119	0.0300	228	0.0288
74	0.0129	126	0.0100	229	0.3942
75	0.0136	143	0.0140	230	0.0672
76	0.0176	154	0.0163	231	0.0158
77	0.1432	155	0.0138	232	0.0109
78	0.0141	156	0.0168	233	0.0266
79	0.0450	157	0.0334	234	0.0121
80	0.0136	158	0.0296	235	0.0173
81	0.0580	159	0.0373	236	0.0199
83	0.1294	166	0.0388	237	0.2613
84	0.0135	187	0.0186	238	0.0330
85	0.0339	189	0.0221	240	0.0232
88	0.0177	190	0.2840	241	0.3797
91	0.0380	191	0.0118	242	0.0396
92	0.0183	194	0.0229	247	0.0123
93	0.0149	196	0.2978		

Table 5.1: Cache sets with mutual information $I(S, \tilde{\boldsymbol{L}}_q) > 0.01$.

Thus this adversary has advantage:

$$Adv(\mathtt{A}_{\mathtt{E}_{\{0,1\}}}) = |P(\mathtt{A}_{\mathtt{E}_{\{0,1\}}} \rightarrow 1 \mid s = 1) - 1/2| = 0.444. \qquad (5.2)$$

Input parameters	
SOM Dimensions:	25
Training samples:	100 x 1 measurements of a $\sim$ 1024 bit key
Challenge sample:	1 x 1024 bit key

Results	
Classification Rate on Training Data:	0.926
Classification Rate on Challenge Data:	0.944
Correct bits recovered:	967/1024

Table 5.2: Results of experiment.

5.3 Conclusion

The significance of the leakage in this section cannot be stressed enough. By observing *one* decryption operation, we are able to extract 967 bits of the secret key.

Although this experiment is performed under ideal conditions, previous work has shown that similar results have been easily expanded to more sophisticated attacks, cf. [2, 5, 8, 18, 26, 48].

However, the attack in its current form has a significant drawback: It relies on the ability to inject code into the decryption function. It is with this in mind that we move on to the next chapter, where we attempt (and fail) to implement a spy that is truly decoupled from the target process.

Chapter 6

Attempting an Asynchronous Attack

In order to make the attack more practical, the next logical step is to try to decouple the spy from the victim. This has been successfully implemented in the literature for simultaneous multi-threading (SMT) platforms, cf. [2]. It is, for the reasons stated below, a very different exercise on ARM.

6.1 Alterations to the Spy

In this setting the spy is no longer called from within the crypto-process, rather it runs in a different process scheduled to run on the same core[1].

As ARM Cortex A9 does not use simultaneous multi-threading (SMT), we rely on the operating system's preemptive scheduler to pause the execution of the crypto-process and start the spy.

6.2 Method

For this experiment, we start two separate processes on the same core, one is repeatedly running the spy, and the other is repeatedly performing RSA-decryptions of random messages using the same simple modular version of the multiply-square algorithm as shown in appendix C, only without the explicit calls to the spy.

The decryption process is started first, then the spy measures N samples, finally the spy exits normally and the decryption process is killed.

[1]We ensure that the processes are forced to run on the same core using the `taskset` utility on Linux.

6.3 Discussion of Results

The results of this experiment are negative, first and foremost stomped by the problem of sampling frequency.

We started our analysis by checking how often we expect our spy to be scheduled, based on the timing of `mod_multiply` and `mod_square`[2]. These timing measurements are an average of 1000000 samples and are listed in table 6.1. All in all, this looks very promising. We expect that, if the spy and decryption process are alternately scheduled at every OS-tick, we should sample around one measurement per multiply or square. This, however, is not the case in practice.

Operation	Time [performance-ticks]
OS tick	36548
`mod_multiply`	38198
`mod_square`	40776

Table 6.1: Timings of operations in an unstressed environment.

We found that in practice, the spy is only scheduled to run after what appears to be every 4–5th OS tick. This effectively means that we are not measuring which operation is happening *right now*, but rather, if there has been a `mod_multiply` run in the last 4–5 operations. Assuming the key is random, this means we are *ideally* only able to learn when there are two or more consecutive 0's in the key. Which might be of *some* use, but not what we set out to measure. There is also added noise from the instructions run by the OS kernel during context-switching in addition to the regular sources observed in the previous experiments.

In addition, we have had a problem with combining the data in the measurements into a coherent whole. The OS sometimes (after a few hundred measurements) decide to leave our process unscheduled for few hundred ticks. This gives rise to the need to patch together partial information from several measurements, and in our experiment the data simply does not align satisfactory using either of a number of techniques ranging from brute force to more sophisticated multiple sequence alignment algorithms [17] adapted from genome research.

Thus, after much deliberation and many dead ends, we conclude that this experiment is a failure, and that an alternate strategy is needed to implement the side-channel with an asynchronous spy on a non-SMT platform.

[2]`mod_multiply` and `mod_square` return $a \cdot b \pmod{p}$ and $a \cdot a \pmod{p}$ respectively.

6.4 Suggested Future Work

The main reason for this experiment's failure is the need for higher resolution sampling. One way to do this, as hinted in the published literature, is to combine I-cache analysis with an exploit on OS scheduling. One such exploit was discovered by Tsafrir in 2007 [19], but this has since been fixed [35]. If, however, it is possible to influence the current Linux scheduler, or that of another operating system, to reliably schedule the spy-process at a sufficient frequency, this attack may still become practical.

Another technique that may give sufficient sampling resolution is to schedule the spy function as an interrupt instead of an actual process. If this can be done reliably it may be even better than leaving it to the OS. Further research into the feasibility and practicality of this approach is required. Unfortunately we are not able to do so in the time allotted for this paper.

6.5 Alternate Asynchronous Attack

A recent work by Yarom and Falkner [48] introduces a variant of the PRIME+PROBE attack used in this paper called FLUSH+RELOAD. It is a more directed attack that utilises shared memory pages between the spy and the victim. Instead of filling the cache with the spy's code as is done in the PRIME-phase of PRIME+PROBE, the spy FLUSHes carefully chosen cache lines from *all* cache levels. Later the spy will attempt to RELOAD each cache line and determine whether it is fetched from main memory or not. FLUSH+RELOAD has since been used several related attacks, cf. [12, 47].

For our purpose, FLUSH+RELOAD (or variants thereof) overcome the challenge of sampling resolution as the higher level caches are shared amongst different physical cores even on ARM. On the other hand this technique comes with a new challenge: The cache eviction instructions require super user privileges, unless the same functionality may be obtained using other legal operations.

Another possibility could be to attempt a PRIME+PROBE attack against the L2 cache which *is* shared amongst different physical cores. The L2 and L3 caches are generally small on embedded devices, which could make this a viable option.

Chapter 7

Countering I-Cache Analysis

This chapter will present the current state of countermeasures against software-based cache side-channels. These countermeasures include changes to both hardware and software and summarise the work in [2, 7, 21, 30, 31, 36, 37, 45].

7.1 Software Transformations

7.1.1 Avoiding Key-Dependent Control Paths

The most intuitive defence against I-cache analysis is to avoid creating key-dependent control flow. The idea is to replace any patterns similar to:

```
if (x != 0)
   var = foo();
/*else
   var = bar();*/ // Optional
```

where x is dependent on the secret input, with a fixed control flow.

This can be enforced in several different ways. One way is to always perform all sensitive calculations, but only use the result for the case where the conditional statement would have branched.

```
res[0] = foo();
res[1] = bar(); // Optional
var = res[ x != 0 ];
```

A downside with to technique is that it will always decrease the performance of the application.

Another similar technique is to replace the functions altogether, and use pre-computed values stored in tables instead. Note, however, that this may make the code susceptible to a data-cache attack, cf. [37, 40].

It is important to stress that this countermeasure is prone to human error if done manually. Even if executed correctly at a point in the code, there is no guarantee that other functions or subroutines are not key-dependent. Sensitive control flow must be avoided at *all* levels of the software stack for this countermeasure to be successful.

7.1.2 Cache Conscious Layout

Cache conscious layout [2] is most easily enforced on the compiler or OS level, but may also be implemented by a developer using alignment flags. Doing so requires accurate knowledge of the cache layout of the target platform. It works by mapping sensitive functions to the same cache sets, thus making it difficult for an adversary to distinguish between them. Note however, that an adversary who is able to measure the cache at a sufficiently high frequency may be able to distinguish between security sensitive functions by analysing the cache access patterns even inside critical functions.

7.1.3 Cryptographic Obfuscation

Another quite recent development that may be used to close the instruction cache side-channel is what is known as program obfuscation. In a recent paper Garg et al. [21] proposed a candidate for *indistinguishability obfuscation.*

Indistinguishability obfuscation is a notion of security stating that given the obfuscated version of two equivalent programs C_0, C_1, an adversary is unable to distinguish between them. This also implies that the adversary is unable learn anything from tracing the program.

A simple argument runs like this: Assume C_0 is an implementation of a program that does not leak information. Whilst C_1 is an equivalent program that does leak information. Let $iO(C_0), iO(C_1)$ be the obfuscated versions of C_0, C_1 respectively. If an adversary is unable to distinguish between $iO(C_0), iO(C_1)$, she cannot learn anything useful about the leakage from C_1, as this knowledge would let her distinguish between the program.

It should be noted that program obfuscation is a field that still require more research. Also, program obfuscation has much more ambitious aims than countering simple side-channel analysis. Nonetheless, we argue that a securely obfuscated program would by necessity also be protected against instruction cache analysis.

7.1.4 Cache Flushing

Acıiçmez et al. [2] also tested full and partial cache flushing as a defence against cache analysis on non-SMT platforms. However, more recent work has shown that cache analysis may be used to target lower level caches as well [48]. Cache flushing may still be a viable defence on platforms utilising exactly one logical core and the countermeasure is implemented at the OS level.

Full cache flushing would invalidate the complete cache on context switching. This would still reduce the performance of all processes on the system, but to a much smaller extent than disabling the cache completely.

Partial cache flushing would have a smaller impact on performance, but comes with other difficulties. The idea is to flush only the cache-sets that may leak information. Thus, the crypto-process must be analysed beforehand to identify critical code-sections. All cache sets mapping a critical sections would then need to be flushed when the crypto-process switches out.

7.2 Hardware Support

7.2.1 Disable Caching

A rather trivial countermeasure is to simply disable/remove the cache altogether. This, however, has a tremendous negative impact on the performance of the whole system. Acıiçmez et al. [2] measured a performance decrease to 0.001 of the baseline[1]. This is not a viable countermeasure in practice.

7.2.2 Cache Partitioning

Another approach, closely related to cache usage in real time systems, cf. [36], is to partition the cache itself so that different processes/security levels map to disjoint sets in the cache [38]. The idea is to deny unprivileged code access to sensitive cache sets, thereby effectively closing the cache side-channel. Given hardware support for physical partitioning of the caches, this could close the cache side-channel even across virtualisation boundaries.

Cache partitioning requires hardware and instruction set support, and is not currently available in general purpose hardware. Another issue with this countermeasure is that partitioning gives each process less cache to work with and will significantly decrease the performance of memory intensive applications.

Furthermore, this countermeasure would not protect against attacks relying on internal cache collisions in the crypto-process, cf. [9].

[1]For RSA-decryptions with a 2048 bit key

7.2.3 Alternate Cache Architectures

The final hardware countermeasures we will present are the *partition locked cache* (PLcache) and *random permutation cache* (RPcache) introduced by Wang and Lee [45] and further improved by Kong et al. [30].

Wang et al. designed the caches to protect against two known data cache attacks against RSA and AES [45]. Kong et al. later showed that adapted attacks could circumvent the protection offered by the PLcache and RPcache [31], but later published amendments that improve the security of both architectures [30].

Both cache architectures require a developer to identify critical sections of an application's address space, whether data or instructions, and handle the loading of these sections using special instructions.

PLcache + Preloading

The idea behind a locked partition cache (PLcache) is similar to the cache partitioning introduced by Percival in [38]. Critical sections are locked in the cache such that these cache lines cannot be evicted. This requires the cache to be large enough to fit the critical sections in the cache.

To give an example, in the case of software implementations of AES it is common to replace the expensive round calculations with precomputed look-up tables. Accesses to these tables are key-dependent and opens a data cache side-channel, cf. [9, 37]. To defend against such attacks the crypto-process would need to preload all the look-up tables, lock them in the cache and perform the encryption(s), before finally unlocking the cache lines.

The preloading is an important step and should not be skipped, as shown by Kong et al. [30, 31].

Implementation Details A PLcache may be implemented by adding one bit L to every cache line to mark whether this line is locked, and replacing the cache logic to that described in figure 7.1 on the facing page.

Limitations There are some limitations to the PLcache that should be mentioned:

1. The PLcache is ill-suited as a secure instruction cache because of the latency involved in preloading, cf. [30].
2. Small caches will result in a non-trivial performance overhead as locking parts of the cache will effectively reduce the cache available to rest of the system.
3. Caches so small that the whole critical set cannot be held in memory at once may still be vulnerable to cache analysis attacks as the preconditions are voided.

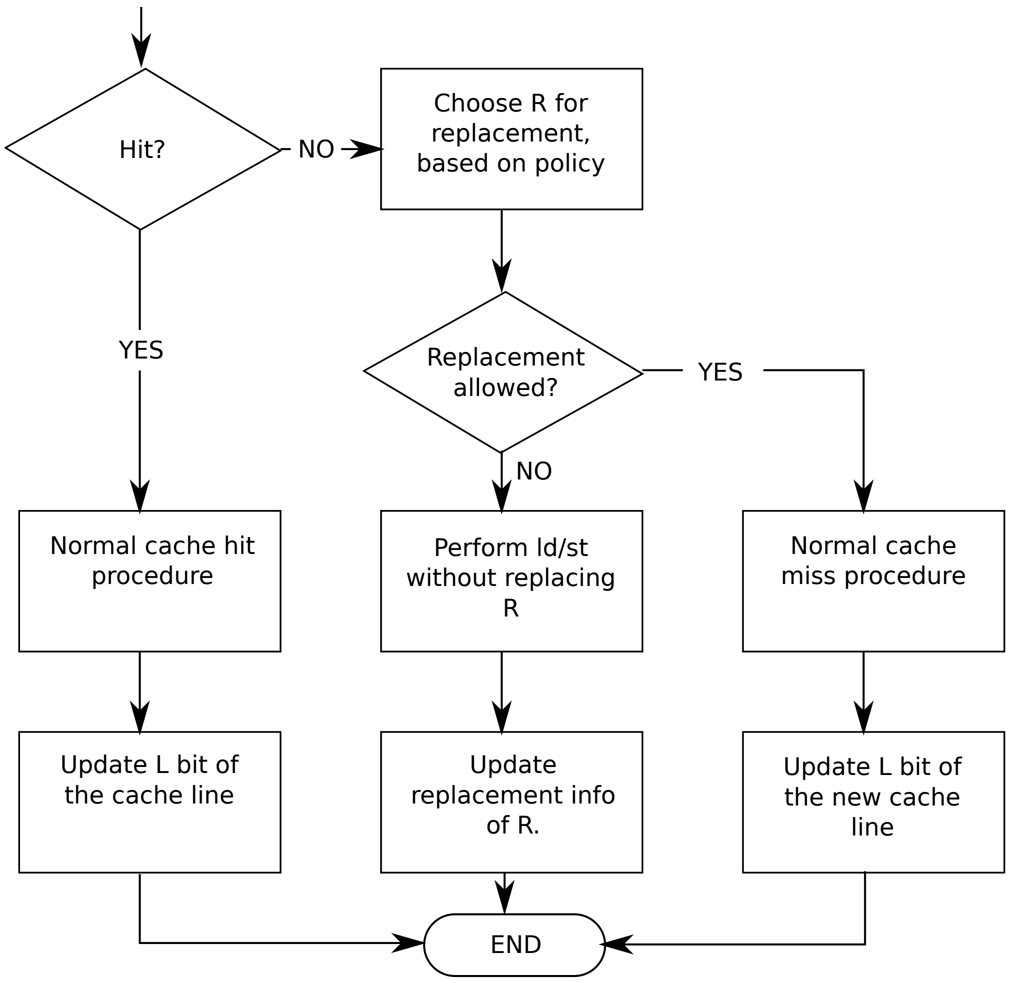

Hit?
NO
Choose R for replacement, based on policy
YES
Replacement allowed?
YES
NO
Normal cache hit procedure
Perform ld/st without replacing R
Normal cache miss procedure
Update L bit of the cache line
Update replacement info of R.
Update L bit of the new cache line
END

Figure 7.1: Cache access handling procedure of PLcache.

4. In order to perform preemptive scheduling, the OS will need to do bookkeeping on the critical sections of each process. It would then have to unlock and reload the critical sections when a sensitive process is rescheduled.
5. Cache line locking should be a privileged operation as a malicious process could potentially lock down the whole cache. Thus choking memory performance.

RPcache + Informing Loads

The random permutation cache (RPcache) functions quite differently from the other countermeasures discussed so far. The idea is to randomise the signal received by an attacker so he can learn nothing from it. We must still identify and mark the critical sections of the code.

Details The most major change to the cache is the addition of two or more indirection tables. The point of this is to add a virtualisation layer on the cache set mapping per process. This cache set mapping is further randomised according to the rules outlined in figure 7.2 on the next page. Each cache line is tagged with the process-id of the owning process and a permutation table index P.

This is best explained by an example:

Example 2. The platform has two separate permutation levels, 0 and 1. A memory location R is marked as sensitive and loaded into the cache in set S. This means that R's permutation table index is one: $P_R = 1$.

Some time later, we attempt to load another memory location D from a different process. This access uses the unprivileged permutation table, $P_D = 0$, and happens to map to the same cache set as R, namely S, where R is next up for eviction.

Now, instead of evicting R, we select a random cache set S' and insert D into S', mercilessly evicting whatever cache line was up for eviction in S'. Furthermore, we swap the mappings of S and S' in the unprivileged permutation table and update the cache lines with $P_X = 0$, so that subsequent accesses to the other entries already in the cache will still result in cache hits.

Furthermore, in order to be secure, it is required that the architecture supports *informing loads* [30, 33], which means that the process performing the load can issue a special load instruction to receive an interrupt if the access resulted in a cache miss. The need for this feature was determined by Kong et al. in [31] where they showed that timing driven attacks could still succeed on a RPcache without informing loads.

Informing loads must be actively used by the sensitive application to ensure that it is not vulnerable to timing attacks, for instance by reloading critical data into the cache, cf. [30].

Another important point to mention is that the use of informing loads may ease implementations of defences against new cache vulnerabilities as new mitigations may be patched into the software initiated by the interrupts.

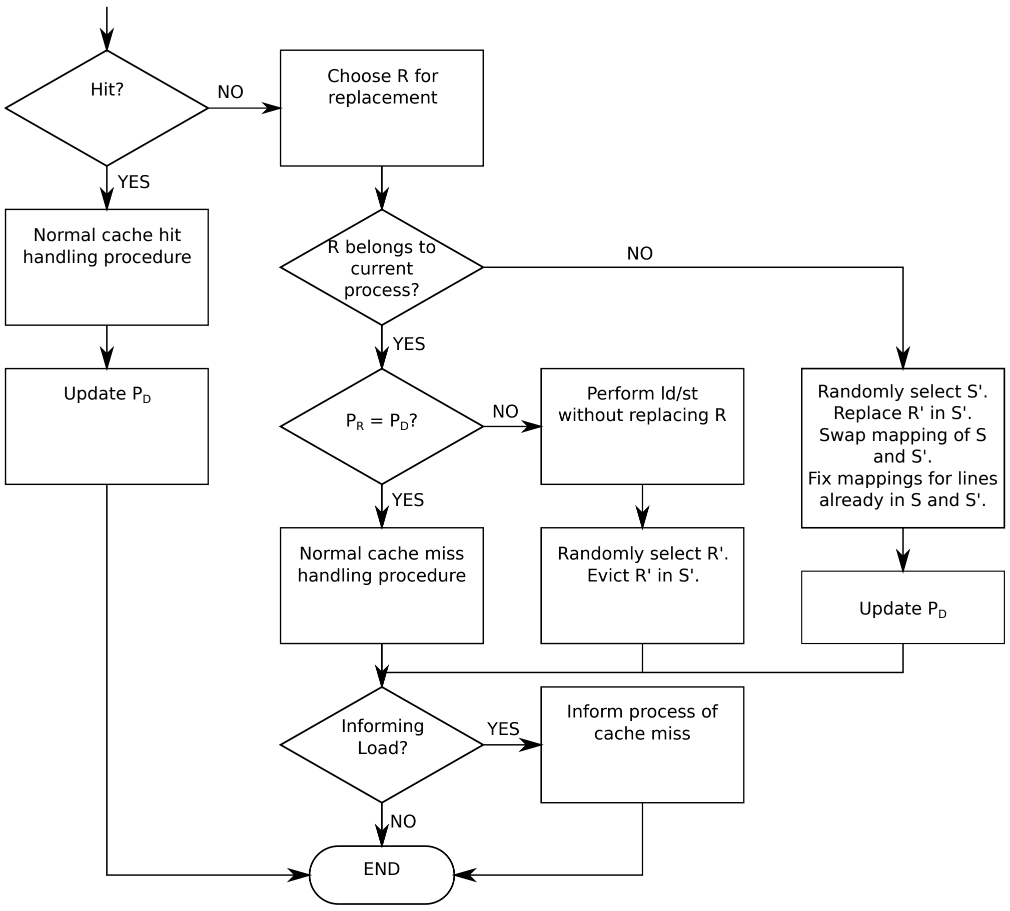

Hit?
NO
Choose R for replacement
YES
Normal cache hit handling procedure
R belongs to current process?
NO
YES
Update P_D
$P_R = P_D$?
NO
Perform ld/st without replacing R
Randomly select S'. Replace R' in S'. Swap mapping of S and S'. Fix mappings for lines already in S and S'.
YES
Normal cache miss handling procedure
Randomly select R'. Evict R' in S'.
Update P_D
Informing Load?
YES
Inform process of cache miss
NO
END

Figure 7.2: Cache access handling procedure of RPcache. Cache line R, R' reside in cache set S, S' respectively.

Chapter 8

Conclusion

This master thesis has managed to reproduce an instruction cache side-channel attack based on several attacks from the literature. With this attack we were able to extract 967 of 1024 bits of an RSA key from observing a single decryption.

Although we were unable to make the spy completely decoupled from the victim, we are convinced that this was due to lack of resources on our part, and not due to the feasibility of the problem.

The ARM architecture does, however, introduce challenges that are not present on x86, namely that timing and cache eviction are privileged operations. In addition, the lack of simultaneous multi-threading makes decoupled attacks more difficult.

Furthermore, it is a key point to stress that the results in this and related papers are still applicable across virtualisation borders. This is of interest to situations where a secure process may share cache with a non-secure process, cf. ARM TrustZone [11]. Great care must be shown to not introduce information leakage at any level of the caches.

8.1 Remaining Work

As previously mentioned, there is still remaining work along the research vector of asynchronous L1-instruction cache attacks on ARM. Ideas that need further consideration include: Using existing OS facilities to schedule the spy more reliably either as an interrupt or as a special process. It should also be investigated whether an unsecured OS may schedule a spy with sufficient sample resolution when running alongside a secure OS in a virtualised environment.

Other remaining work include implementing more sophisticated attacks, as for instance that of [2]. However, it should be noted that this would probably not introduce any new results. As we show in this paper, there is indeed leakage.

Thus, known attacks already demonstrated on x86 should also be applicable to ARM platforms assuming the issue of sampling resolution can be addressed.

The last remaining work that we will mention is further analysis into the efficiency and power-per-area of the alternate cache designs presented in section 7. More specifically the PLcache and RPcache, which seem to be the most promising countermeasures against cache side-channel attacks.

8.2 Acknowledgements

The author would like to thank the Base team at ARM's offices in Trondheim for their insight into the Linux kernel and for their help on various ARM related questions. The work on this thesis has been greatly accelerated because of their help.

Secondly, I would like to thank my supervisor, Harald Hanche-Olsen for healthy discussions and advice, and for taking his time to help me get started in the cases where I have ventured into fields I have had little to no prior knowledge.

Finally, I would not have been able to finish this master thesis on time if not for my wonderful wife Sarah, whose unending support and solitary parenting throughout the summer gave me the time and energy to finish this master thesis. *Factum est.*

Bibliography

[1] Onur Acıiçmez. Yet another microarchitectural attack:: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18. ACM, 2007.

[2] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 110–124. Springer, 2010.

[3] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In *Cryptography and Coding*, pages 185–203. Springer, 2007.

[4] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Topics in Cryptology–CT-RSA 2007*, pages 225–242. Springer, 2006.

[5] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320. ACM, 2007.

[6] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on aes. *IACR Cryptology ePrint Archive*, 2006:138, 2006.

[7] Onur Acıiçmez and Çetin Kaya Koç. Microarchitectural attacks and countermeasures. In *Cryptographic Engineering*, pages 475–504. Springer, 2009.

[8] Onur Acıiçmez and Werner Schindler. A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In *Topics in Cryptology–CT-RSA 2008*, pages 256–273. Springer, 2008.

[9] Onur Acıiçmez, Werner Schindler, and Çetin K Koç. Cache based remote timing attack on the aes. In *Topics in Cryptology–CT-RSA 2007*, pages 271–286. Springer, 2006.

[10] Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*, pages 80–91. IEEE, 2007.

[11] ARM. Arm security technology building a secure system using trustzone® technology. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html` - Accessed 25.08.2014., 2009.

[12] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. "ooh aah... just a little bit": A small amount of side channel can go a long way. *IACR Cryptology ePrint Archive*, 2014:161, 2014.

[13] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In *Advances in Cryptology—CRYPTO'98*, pages 1–12. Springer, 1998.

[14] Dan Boneh et al. Twenty years of attacks on the rsa cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999.

[15] Billy Bob Brumley and Risto M Hakala. Cache-timing template attacks. In *Advances in Cryptology–ASIACRYPT 2009*, pages 667–684. Springer, 2009.

[16] Mauro Brunato. Advanced business intelligence techniques 11: Self-organizing maps. `https://www.youtube.com/watch?v=4Z7i6HSx0Bc` accessed 4th August 2014., July 2012.

[17] CBRC and AIST. Mafft multiple sequence text alignment. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt` - Accessed 10.09.2014., 2014.

[18] CaiSen Chen, Tao Wang, Yingzhan Kou, XiaoCen Chen, and Xiong Li. Improvement of trace-driven i-cache timing attack on the rsa algorithm. *Journal of Systems and Software*, 2012.

[19] Tsafrir D., Etsion Y., and Feitelson D.G. Secretly monopolizing the cpu without superuser privileges. In *Proceedings of the 16th USENIX Security Symposium (SECURITY 2007)*, pages 239–256. USENIX Association, 2007.

[20] Ulrich Drepper. What every programmer should know about memory. `http://www.akkadia.org/drepper/cpumemory.pdf`, Nov 2007.

[21] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 40–49. IEEE, 2013.

[22] Daniel Genkin, Itamar Pipman, and Eran Tromer. Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs. 2014.

[23] Allen Gersho and Robert M. Gray. *Vector quantization and signal compression*. Kluwer Academic Publishers, Massachusetts, US, 1992.

[24] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In *Cryptographic Hardware and Embedded Systems–CHES 2008*, pages 426–442. Springer, 2008.

[25] Philipp Grabher, Johann Großschädl, and Daniel Page. Cryptographic side-channels from low-power cache memory. In *Cryptography and Coding*, pages 170–184. Springer, 2007.

[26] NA Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography*, 23(3):283–290, 2001.

[27] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.

[28] Teuvo Kohonen. Improved versions of learning vector quantization. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 545–550. IEEE, 1990.

[29] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[30] Jingfei Kong, Onur Aciicmez, J-P Seifert, and Huiyang Zhou. Architecting against software cache-based side-channel attacks. *Computers, IEEE Transactions on*, 62(7):1276–1288, 2013.

[31] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM workshop on Computer security architectures*, pages 25–34. ACM, 2008.

[32] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *Computer Aided Verification*, pages 564–580. Springer, 2012.

[33] Margaret Martonosi, Michael D Smith, Todd C Mowry, and Mark Horowitz. Informing memory operations: Providing memory performance feedback in modern processors. In *Computer Architecture, 1996 23rd Annual International Symposium on*, pages 260–260. IEEE, 1996.

[34] Silvio Micali and Leonid Reyzin. Physically observable cryptography. In *Theory of Cryptography*, pages 278–296. Springer, 2004.

[35] Ingo Molnar. Completely fair scheduler – introduced in linux kernel v2.6.23. `http://mafft.cbrc.jp/alignment/software/textcomparison.html` - Accessed 10.09.2014., Oct. 2007.

[36] Frank Mueller. Compiler support for software-based cache partitioning. *ACM Sigplan Notices*, 30(11):125–133, 1995.

[37] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology–CT-RSA 2006*, pages 1–20. Springer, 2006.

[38] Dan Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptology ePrint Archive*, 2005:280, 2005.

[39] Colin Percival. Cache missing for fun and profit. BSDCan, 2005.

[40] Chester Rebeiro and D Mukhopadhay. Boosting profiled cache timing attacks with a priori analysis. 7(6), December 2012.

[41] Claude Elwood Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, XXVII(3):379–423, July 1948.

[42] François-Xavier Standaert, Tal G Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In *Advances in Cryptology-EUROCRYPT 2009*, pages 443–461. Springer, 2009.

[43] Colin D Walter. Montgomery exponentiation needs no final subtractions. *Electronics letters*, 35(21):1831–1832, 1999.

[44] Colin D Walter and Susan Thompson. Distinguishing exponent digits by observing modular subtractions. In *Topics in Cryptology—CT-RSA 2001*, pages 192–207. Springer, 2001.

[45] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM, 2007.

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

[47] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[48] Yuval Yarom and Katrina E Falkner. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. *IACR Cryptology ePrint Archive*, 2013:448, 2013.

Appendix A

Using Performance Timing on Cortex A9

The code below is adapted from stackoverflow.com[1].

In order to enable access to the cycle count register, the following code must be run from an elevated privilege level (Kernel space or above) once on each CPU core.

```
// program the performance-counter control-register:
asm volatile ("MCR p15, 0, %0, c9, c12, 0\t\n" :: "r"(0x00000017));

// Enable the CCNTR register:
asm volatile ("MCR p15, 0, %0, c9, c12, 1\t\n" :: "r"(0x80000000));

// Clear overflow status flag:
asm volatile ("MCR p15, 0, %0, c9, c12, 3\t\n" :: "r"(0x80000000));

// Enable user-mode access to the performance counters
asm volatile ("MCR p15, 0, %0, C9, C14, 0\n\t" :: "r"(1));

// Disable counter overflow interrupts (just in case)
asm volatile ("MCR p15, 0, %0, C9, C14, 2\n\t" :: "r"(0x80000000));
```

[1]http://stackoverflow.com/a/3250835/2430032

Once access to the register is enabled any process can read the current cycle count using follwing code.

```
 inline unsigned int get_cyclecount()
 {
  unsigned int value;
  // Read CCNT Register
  asm volatile ("MRC p15, 0, %0, c9, c13, 0\t\n": "=r"(value));
  return value;
}
```

Appendix B

Assembly Code of The Spy

Below is an excerpt from the assembly code for the spy process written in ARM assembly. It is automatically generated by a Python script that can be found in electronic appendix, available on DAIM. The header file *asm_ defines.h* defines the constant `CACHE_SETS`.

```
/*This file was generated with parameters:
    Cache size: 32768 bytes,
    Cache line size:
    32 bytes and cache associativity: 4*/

/* AUTOMATICALLY GENERATED */
#include "asm_defines.h"

.text
.globl spy_loop_abs

/*void spy_loop_abs(  abs_t * raw_timings, unsigned long N)*/

spy_loop_abs:

    /* R0 holds argument raw_timings*/
    /* R1 holds argument N which is number of rounds*/
    /* R2 holds current measurement */
    /* R3 holds previous measuremnt */

    /* PROLOG */
    /* Start */
    MOV R2, #CACHE_SETS
    MUL R1, R1, R2

    MOV R2, #RAW_T_SIZE
    MLA R1, R1, R2, R0  /* R1 = R1*R2 + R0 */

    /* Save first timing. */
    MRC p15, 0, R3, c9, c13, 0
.balign 8192
L00000:
    B L00256
    .rept 7
        NOP
    .endr
L00001:
    B L00257
    .rept 7
        NOP
    .endr
L00002:
    B L00258
    .rept 7
        NOP
    .endr
L00003:
    B L00259
    .rept 7
        NOP
    .endr
/*(...snip...)*/
L01021:
    MRC p15, 0, R2, c9, c13, 0
    CMP R2, R3
    SUBLE R3, R3, R2
    SUBGT R3, R2, R3
    STR R3, [R0], #4
    MRC p15, 0, R3, c9, c13, 0
    B L00254
    NOP
L01022:
    MRC p15, 0, R2, c9, c13, 0
    CMP R2, R3
    SUBLE R3, R3, R2
    SUBGT R3, R2, R3
    STR R3, [R0], #4
    MRC p15, 0, R3, c9, c13, 0
    B L00255
    NOP
L01023:
    MRC p15, 0, R2, c9, c13, 0
    CMP R2, R3
    SUBLE R3, R3, R2
    SUBGT R3, R2, R3
    STR R3, [R0], #4
    MRC p15, 0, R3, c9, c13, 0
    CMP R0, R1    /* This is ok, since we are only
                     comparing for whole cache sets.*/
    BLO L00000
    B END
END:

    /* EPILOG */

    MOV pc, lr
```

Appendix C

Multiply-Square Algorithm

Below is the actual C-code for the implementation of RSA used in the experiments of chapter 5 and 6. *types.h* defines the type `raw_t`, *asm_defines.h* defines the constant `CACHE_SETS`.

simple-rsa.h

```c
#pragma once

#include <openssl/bn.h>
#include "types.h"

#ifdef __cplusplus
extern "C" {
#endif

void mod_exp( BIGNUM *ret, const BIGNUM *base, const BIGNUM *exponent,
                const BIGNUM *mod , raw_t *raw_data);
void rsa_decrypt( BIGNUM *plaintext, const BIGNUM *ciphertext,
                const BIGNUM *n, const BIGNUM *d , raw_t *raw_data);

#ifdef __cplusplus
};
#endif
```

simple-rsa.c

```c
#include "simple_rsa.h"
#include "asm_defines.h"
#define RESET_SAMPLES 25
static raw_t   resetbuffer[RESET_SAMPLES*(CACHE_SETS)];

extern void spy_loop_abs( raw_t * raw_timings, unsigned long N);

void rsa_decrypt( BIGNUM *plaintext, const BIGNUM *ciphertext,
                const BIGNUM *n, const BIGNUM *d , raw_t *raw_data) {
        mod_exp( plaintext, ciphertext, d, n , raw_data);
}

void mod_exp( BIGNUM *ret, const BIGNUM *base, const BIGNUM *exponent,
                const BIGNUM *mod , raw_t *raw_data) {
        int nbits, i;
        BIGNUM *multiplier;
        BN_CTX *ctx;

        multiplier = BN_dup( base );
        BN_one(ret); // OK, even if base == ret, as we made a copy above.

        ctx = BN_CTX_new();
        BN_CTX_init(ctx);

        i = 0;
        nbits = BN_num_bits(exponent);
        while ( i < nbits ) {
                // Reset ownership
                spy_loop_abs( resetbuffer, RESET_SAMPLES );

                if ( BN_is_bit_set(exponent, i) ) {
                        BN_mod_mul(ret, ret, multiplier, mod, ctx);
                }

                BN_mod_sqr(multiplier, multiplier, mod, ctx);
                // Measure
                spy_loop_abs( raw_data + i*(CACHE_SETS), 1 );
                ++i;
        }
        BN_clear_free(multiplier);
        BN_CTX_free(ctx);
}
```