USING AUTOCORRELATIONS TO DETECT POTENTIAL SIDE-CHANNELS

Caroline Butschek

Thesis of 30 ECTS credits
Master of Science (M.Sc.) in Computer Science

February 2018
Using Autocorrelations to Detect Potential Side-Channels

by

Caroline Butschek

Thesis of 30 ECTS credits submitted to the School of Computer Science at Reykjavík University in partial fulfillment of the requirements for the degree of Master of Science (M.Sc.) in Computer Science

February 2018

Supervisor:
Marcel Kyas, Supervisor
Assistant Professor, Reykjavík University, Iceland

Examiner:
Ýmir Vigfússon, Examiner
Associate Professor, Emory University, USA

Jon Gudnason, Examiner
Associate Professor, Reykjavík University, Iceland
Abstract

Side-channel attacks like timing attacks are a severe threat towards cyber security. Although this type of attack has been known for more than 10 years, exploits based on timing attacks are still published. The aim of this work is to find a simple testing method for crypto libraries, that shows potential vulnerabilities. Therefore a precise measurement method is crucial. Firstly experiments with Python benchmarking tools have been performed using a constant time AES implementation. The measurement has been repeated using the method proposed by Intel, that recommends the programming language C, specific instructions and the kernel mode. This measurement method performed better than the one in Python. Furthermore the autocorrelation method and the Ljung-Box test have been applied to measurement results of the tests performed in C, but no large autocorrelations that indicate a timing leak were found. The measurement method was then applied to OpenSSLs RSA decryption function, which leaks exploitable timing variances, if blinding is disabled. Although higher autocorrelations were found within this experiment, the results were so heavily affected by noise, that it was difficult to determine if they came from the timing leak or from system noise. Therefore the goal to develop a simple testing method has not been reached.
Using Autocorrelations to Detect Potential Side-Channels

Caroline Butschek

Thesis of 30 ECTS credits submitted to the School of Computer Science at Reykjavík University in partial fulfillment of the requirements for the degree of Master of Science (M.Sc.) in Computer Science

February 2018

Student:

Caroline Butschek

Supervisor:

Marcel Kyas

Examiner:

Þýmir Vigfússon

Jon Gudnason
The undersigned hereby grants permission to the Reykjavík University Library to reproduce single copies of this Thesis entitled Using Autocorrelations to Detect Potential Side-Channels and to lend or sell such copies for private, scholarly or scientific research purposes only. The author reserves all other publication and other rights in association with the copyright in the Thesis, and except as herein before provided, neither the Thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author’s prior written permission.

Date

Caroline Butschek
Master of Science
# Contents

Contents .... xi

List of Figures .... xiii

List of Tables .... xv

1 Introduction .... 1
   1.1 Goal of the thesis .... 2

2 Related Work .... 5
   2.1 Kochers Timing Attack on an Implementation of Diffie-Hellman .... 5
   2.2 Bernstein’s Cache Timing Attack on AES .... 6
   2.3 Brumley and Boneh’s Remote Timing Attack on RSA .... 7
   2.4 Morgan and Morgan’s Classification of Potential Timing Leaks .... 8
   2.5 Kaufmann et al. Exploit an Elliptic Curve Diffie-Hellman Key Exchange Implementation .... 9

3 Experiments .... 11
   3.1 Hardware .... 11
   3.2 Measurement of Execution Times in Python .... 12
      3.2.1 Python .... 12
      3.2.2 Python Extensions .... 12
      3.2.3 PyCrypto .... 13
      3.2.4 Measurement Methods in Python .... 18
   3.3 Measurement in Python with Different Schedulers on Different Cores .... 19
      3.3.1 Setup .... 19
      3.3.2 Results .... 20
      3.3.3 Discussion and Conclusion .... 23
   3.4 Execution Times of Different Keys and Different Blocks in Python .... 24
      3.4.1 Setup .... 24
      3.4.2 Results .... 25
      3.4.3 Discussion and Conclusion .... 25
   3.5 Measurement in C .... 25
      3.5.1 The Kernel Method suggested by Intel .... 26
      3.5.2 Kernel Programming .... 27
   3.6 Measurement with One Key and One Block .... 28
      3.6.1 Setup .... 28
      3.6.2 Results .... 29
      3.6.3 Discussion and Conclusion .... 32
List of Figures

1.1 Problem Setting .......................................................... 2
3.1 Overview of a Simple Extension ........................................ 13
3.2 Overview over the AES implementation in PyCrypto .................... 14
3.3 Calling a new instance of a Cypher .................................... 16
3.4 Encrypt Blocks ............................................................ 17
3.5 Number of Executions in Minimal Time for \texttt{time()} .............. 22
3.6 Trace from \texttt{time().time()} ........................................ 23
3.7 Trace of 250 000 measured Encryptions .............................. 29
3.8 Trace of the first and the last 250 Encryptions - First Setup ....... 30
3.9 Trace of the first 250 Encryptions - Second Setup .................. 30
3.10 Trace of the first 250 Encryptions - Second Setup .................. 31
3.11 Distribution of Encryption Times for all Setups .................... 32
3.12 Printing measurement results out immediately with \texttt{printk} ....... 33
3.13 The Sine Signal .......................................................... 34
3.14 Noise .................................................................. 34
3.15 Signal Without Noise ..................................................... 34
3.16 Signal With Noise ......................................................... 34
3.17 Biased acf ................................................................ 36
3.18 Unbiased acf ............................................................... 36
3.19 The $\chi^2$ distribution for $n \in \{1, 2, 4, 6 \}$ [32] ................. 37
3.20 Autocorrelations for four additions - 400 iterations in total - only the first 20 iterations are presented ............................... 40
3.21 Autocorrelations for five additions - 400 iterations in total - only the first 20 iterations are presented ............................... 41
3.22 Autocorrelations for six additions - 400 iterations in total - only the first 20 iterations are presented ............................... 41
3.23 Autocorrelations for seven additions - 400 iterations in total - only the first 20 iterations are presented ............................... 41
3.24 Autocorrelations of Different Keys - 20 Iterations ................. 44
3.25 The last 20 Iterations of Key 18 ........................................ 44
3.26 Autocorrelations of Key 18 - 400 iterations in total - only the first 20 iterations are presented ............................... 45
3.27 Autocorrelations of Key 18 - 400 iterations in total (shuffled) - only the first 20 iterations are presented ............................... 45
3.28 Autocorrelations of Different Blocks - 20 Iterations ............... 45
3.29 Autocorrelations of Different Blocks - 400 iterations in total - only the first 20 iterations are presented ............................... 46
3.30 Autocorrelations of Different Blocks - 400 iterations in total (shuffled) - only the first 20 iterations are presented .......................... 46
3.31 Autocorrelations of Different Keys - 20 Iterations .......................... 48
3.32 Signal of Key 0 .......................... 48
3.33 Autocorrelations of Different Keys - 400 iterations in total (shuffled) - only the first 20 iterations are presented .......................... 49
3.34 Autocorrelations of Different Blocks - 20 Iterations .......................... 49
3.35 Autocorrelations of Different Blocks - 400 iterations in total (shuffled) - only the first 20 iterations are presented .......................... 50
3.36 Execution times of Key 0 Block 8 in User Mode (blue) and in Kernel Mode (red) .......................... 52
3.37 Autocorrelations of Different Keys - 400 iterations in total (shuffled) - only the first 20 iterations are presented .......................... 53
3.38 Autocorrelations of Different Blocks - 400 iterations in total (shuffled) - only the first 20 iterations are presented .......................... 54
3.39 Autocorrelations of Different Keys - no Blinding (400 iterations in total - shuffled - only the first 30 autocorrelations are shown) .......................... 56
3.40 Autocorrelations of Different Blocks - no Blinding (400 iterations in total - shuffled - only the first 30 autocorrelations are shown) .......................... 56
3.41 Timing traces of key 0 and key 2 .......................... 58
3.42 Autocorrelations of Different Blocks - no Blinding (400 iterations in total - not shuffled - only the first 30 iterations are shown) .......................... 58
3.43 Autocorrelations of unblinded Keys and Blocks (400 iterations in total - shuffled - only the first 30 iterations are shown) .......................... 59
3.44 Autocorrelations of unblinded Keys and Blocks (400 iterations in total - shuffled - only the first 30 iterations are shown) .......................... 59
3.45 Timings of Key 0 Block 2 .......................... 59
3.46 Trace of Key 4 .......................... 60
3.47 Autocorrelations of unblinded keys and blocks (200 iterations in total - shuffled - only the first 30 iterations are shown) .......................... 61
3.48 Autocorrelations of unblinded keys and blocks (200 iterations in total - not shuffled - only the first 30 iterations are shown) .......................... 61
3.49 Autocorrelations of blinded keys (200 iterations in total - shuffled - only the first 30 iterations are presented) .......................... 61
3.50 Autocorrelations of blinded keys (200 iterations in total - not shuffled - only 30 iterations are shown) .......................... 62
3.51 Autocorrelations of blinded blocks (200 iterations in total - shuffled - only the first 30 iterations are shown) .......................... 62
3.52 Autocorrelations of blinded blocks (200 iterations in total - not shuffled - only the first 30 iterations are shown) .......................... 62
3.53 Trace of Key 7 (Last 20 Iterations - blinded) .......................... 63
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Clock Resolutions</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Test Scenarios for Time Measurements in Python</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Average Encryption Times for on a Single Core with noise</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Average Encryption Times of <code>time()</code></td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Average runtime of fastest Execution</td>
<td>23</td>
</tr>
<tr>
<td>3.6</td>
<td>Statistics of Different Setups</td>
<td>31</td>
</tr>
<tr>
<td>3.7</td>
<td>Ljung-Box Test results for autocorrelations with constant blocks</td>
<td>47</td>
</tr>
<tr>
<td>3.8</td>
<td>Ljung-Box Test results for autocorrelations with constant keys</td>
<td>47</td>
</tr>
<tr>
<td>3.9</td>
<td>Ljung-Box-Test results for autocorrelations with constant blocks</td>
<td>50</td>
</tr>
<tr>
<td>3.10</td>
<td>Ljung-Box-Test results for autocorrelations with constant keys</td>
<td>51</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The Federal Office for Information Security in Germany (BSI) defines a cyber attack to be an action that uses IT instruments with the aim to compromise the information security of one or more IT systems within or through the cyber room partly or completely [1]. The cyber room is a worldwide virtual room, which consists of all IT-systems, that potentially can be or are part of a network. Its foundation is the public internet, which can be extended by further networks [1].

Roughly every second company in Germany has been attacked during the years 2013 and 2014, causing costs of more than 50 billion Euros per year according to the German inter-trade organization Bitkom [2]. The Ponemon Institute reports average total costs of cyber attack in Germany to be 7.5 million Dollars, for the UK to be 6.3 million dollars and in the US to be 15,4 million dollars for their samples [3]. In fact it is difficult to determine the real costs, since not all organization notice them, track them or report them. Recent developments like the Internet of Things increase the amount of devices connected to the Internet significantly and therefore they introduce an even larger opportunity to cause damage.

Countermeasures to protect IT systems from attacks include mathematical methods like encryption and software verification. However even the strongest mathematical model does not help, when it its implementation unintentionally leaks information on side channels. A side channel attack is an attack on a cryptographic system, which exploits the results of physical measurements, eg. energy consumption, electromagnetic radiation or execution times of a specific operation in order to gain insight into sensitive data [1]. The BSI classifies this type of attack to be highly relevant for the practical security of any IT system [1].

Several types of side channel attacks exist. Timing attacks observe input- or key-dependent variances of execution times and apply statistical methods and previous knowledge about the implementation to recover the secrets. This type of attack can be applied to local or remote machines and also to devices like smart cards [4]. Potential sources for these timing variances are optimizations to bypass unnecessary operations, branching or conditional statements, cache hits and processor instructions which do not run in a fixed amount of time like multiply and divide. The first attack of this type was published by Kocher in 1996 [5].

Cache-based attacks gather information from the cache, which is shared between two or more processes or between virtual machines. Attackers can watch the cache access pattern from the another process or the other virtual machine and use them in order to recover secrets [4]. Most of the current cache attacks are based on the Flush+Reload attack published by
Yarom et al in 2014 [6], eg. cache-bleed [7] or the recently published attack on sliding windows [8]. The main concept is to remove a specific entry from the cache, and to measure the time, that it takes for the observing process to access this entry again. If the content can be accessed fast, then the entry has also been accessed by the process or vm under observation.

Fault attacks exploit known computational errors, which are provoked for instance by using corrupted inputs. Error message attacks exploit error messages from some specific implementations. Further attack types include power analysis, electromagnetic radiation, acoustic signals and visible light [4].

Writing secure software is challenging. One major problem is shown in Figure 1.1. In order to protect some secret data, the programmer might use or write an encryption library. The code is compiled, and eventually optimized by the compiler. Compilers like the gcc 1 apply a lot of features, optimizations and complex algorithms to the source code, therefore is possible, that the program unintentionally leaks information after compilation. The effects of small changes in the source and different compilation parameters have been studied briefly by Boneh and Brumley in 2005 [9]. Because of their complexity, compilers are often blackboxes to the average programmer and they often do not really know what happens to the source code.

A compiled program is usually executed on specific hardware, like a specific processor with its corresponding cache and a specific network. This hardware might also introduce information leaks. On shared hardware, the attacker can then use a passive attack in order to gather secret information. For this reason, it is not enough to check only the source code. Instead it is important to test an application on its target platform too.

![Figure 1.1: Problem Setting](image)

### 1.1 Goal of the thesis

The goal of this work is to find a simple testing method, which can be applied to the compiled code, which is running on the target hardware. This method should allow the programmer to find potential leaks, that may enable attackers to execute a timing attack. The challenge of this task is the noise caused by the operating system and by other processes that share the hardware. This noise affects the measurement results massively. Furthermore the measurement method itself influences the results.

---

1https://gcc.gnu.org/
First of all, several time measurement techniques are explored in order to get an idea about the performance of time measurement tools. The interpreted high-level language Python and the more low level language C serve as examples. Besides some simple statistical methods like the minimum, mean, median and maximum, the autocorrelation function and the Ljung-Box test are applied to the measurement results in order to find pattern, which arise from the execution times of different encryptions or decryptions. These methods are applied to constant time algorithms like AES and to non-constant time algorithms like RSA. Furthermore this work also explores the impact of different scheduling strategies like FIFO, the attachment of test programs to specific processor cores or the execution of tests in kernel mode on the amount of noise.
Chapter 2

Related Work

This chapter presents some selected timing attacks, which have been published during the last years.

2.1 Kochers Timing Attack on an Implementation of Diffie-Hellman

The first timing attack was presented by Paul Kocher in 1996 [5]. It was based on the observation that conditional operations, whose execution depends on the secret key, result in measurable time differences and leak information about the secret key [5].

The cipher text $R$ is given by the function $R = y^x \mod n$, where $y$ is a message which has been eavesdropped by the attacker, $x$ is the secret key of length $w$ and $n$ is a publicly known modulus. The algorithm to calculate $R$ from the key bits of $x$ is given by listing 2.1. in order to calculate $R$, the algorithm iterates over the key bits of key $x$. The function $R_k = (s_k \cdot y_i) \mod n$ in line 5 is only executed, if bit $k$ of key $x$ is set to 1 within that loop. All other operations in the loop are always executed.

Listing 2.1: Algorithm to calculate $R = y^x \mod n$

```
1  Let $s_0 = 1$.
2  // Iterate over the key bits:
3  For $k = 0$ upto $w - 1$:
4      If (bit $k$ of $x$) is 1 then
5          Let $R_k = (s_k \cdot y) \mod n$.
6      Else
7          Let $R_k = s_k$.
8      Let $s_{k+1} = R_k^2 \mod n$
9  EndFor.
```

Kocher observed, that the modular multiplication is very slow for some pairs of the bit value $s_k$ and the secret message $y$, while other pairs are a lot faster. If a slow modular multiplication pair always results in slow total execution times, then the key bit is probably set to 1, whereas fast total execution times indicate, that the key bit is probably set to 0. Furthermore correctly guessed bits of a previous loop reduce the variance of the total execution time, while incorrectly guessed bits increase it [5].
Kocher used the RSAREF toolkit on a Pentium MSDOS system to observe different modular multiplication times in the RSAREF Diffie Hellman demonstration program. Within a loop iteration, RSAREF processes two modular squaring operations and an optional modular multiplication operation, whose execution depends on the two exponentiation bits that serve as inputs for that loop [5].

The following assumptions were made about the attacker:

- The message \( y \) and the modulus \( n \) are known.
- The design of the target system is known.
- The execution time of \( R_k = (s_k \cdot y_i) \mod n \) is known.
- The total execution time \( t \) of \( R = y_i^k \mod n \) is known.

Kocher measured the execution times of 1 million modular multiplications from actual modular exponentiation operations with random input values for \( s_k \) and \( y \) and a fixed \( n \). Outliers with extremely high execution times, which were above a specific threshold were abandoned. Furthermore he timed the execution of 5000 modular exponentiations, that consisted of 20 groups with 250 timings each. Since the variances should decrease for correctly guessed bits, he guessed the values of two exponent bit, subtracted the corresponding iteration time of the loop from the total execution time and calculated, if the variances of the 250 samples in each group increased or decreased. This way he recovered the whole key.

Although Kocher states, that a precise measurement method is crucial, he does not give any hint about how he measured the execution times [5]. However even if he had given more information about precise measurements, the method would have probably been deprecated by now. This is the reason why some experiments described within this thesis also cover the characteristics of some selected current measurement methods.

Kocher warned, that methods similar to the one he presented could be used to recover RSA and DSS keys [5].

2.2 Bernstein’s Cache Timing Attack on AES

Kocher did not believe, that constant time encryption algorithms are a secure solution to timing attacks. In 2005 Bernstein published a timing attack where he recovered the key of an AES implementation, that theoretically should have run in constant time [10]. This confirmed Kocher’s concerns. The observed execution time variances are caused by S-boxes. S-boxes are lookup tables, which contain fixed values that are used to scramble the values of message and key. Depending on these input values, a value from the S-box is picked and used for further calculations. This step causes exploitable timing variances.[10].

For his experiment, Bernsteins used a client-server-setup. The server ran on a 850 MHz Pentium III desktop computer with the operating system FreeBSD 4.8. It listened for UDP messages on a specific port. If a packet arrived, it used the first 16 bytes as nonce and saved...
the current timestamp. It encrypted the input data and sent back the nonce, a precomputed scrambled zero and the timestamp of the incoming packet and the timestamp that was created after the encryption [10].

The client sent random 400, 600 or 800 byte packets. For analysis, Bernstein chose a fixed value for a byte and calculated the average total execution times of these messages. In addition he calculated the average total execution times of messages containing random bytes only. These average two total execution times significantly differed from each other [10].

To correlate the data, Bernstein used the formula \( \sum_{j=0,1,...,255} t(j)u(i\oplus j) \), where \( i = 0,1,...,255 \) are byte values of a key and \( j \) are the byte values of the nonce. The test to measure the execution times was split into two parts, each containing the measurements of \( 2^{\frac{1}{2}} \) timings. \( t(j) \) represents the average total execution times for a specific value of a specific byte minus the average total execution time of all values for that byte in the first part of the test. \( u(j) \) is calculated the same way using the results from the second part of the test. High correlations indicate, that \( i \) is a possible key byte for \( j \) [10].

By collecting a lot of samples and varying the packet size in addition to some guessing, Bernstein recovered the full key, that scrambled the zero, which the server always sent in its reply [10].

Bernstein used the cycle counter from the processor in order to obtain a high resolution time stamp. The work described in this thesis confirms, that this method is one of the best approaches to measure differences of execution times.

## 2.3 Brumley and Boneh’s Remote Timing Attack on RSA

A second attack from 2005 was presented by Brumley and Boneh [9]. It was performed against OpenSSL 0.9.7, where RSA blinding is disabled by default. They used a 2.4 GHz Pentium 4 processor with one GB of RAM, which ran RedHat Linux 7.3. They used gcc 2.6 provided by RedHat. The keys, which they guessed, were generated at random using OpenSSL’s key generation routine. The server, which they used on their local machine was a simple TCP server, that received and ASCII string, which it then converted to OpenSSL’s multi-precision representation. After the encryption is finished, the server returns a 0 to the client. In the local setup, the client measured the time after sending the message and after receiving the message using RDTSC and CPUID. They also successfully repeated this experiment with a server on a second machine, which was connected via a 10/100Mb Hawking switch [9].

A second experiment they performed used a server running Apache 1.3.27+mod_SSL 2.8.12 and stunnel 4.04, which was commonly used on web servers at that time. The SSL client used the same measurement technique as previously described, but in this case it measures the the time from sending the CLIENT-KEY-EXCHANGE to the reply received from the server. The server was on a separate machine located half a mile away, which was connected via three routers and a number of switches from the campus network. This attack has also been successful [9].

The timing leaks they exploit arise from two sources. The first source is the Montgomery reduction, which was firstly identified by Schindler [11]. Let \( g^d \mod q \) be an exponentiation.
The number of extra reductions depends on the distance between \( g \) and \( q \). The closer \( g \) is to \( q \) or to a multiple of \( q \), the more extra reductions are executed. However if \( g \) is equal to \( q \) or equal to a multiple of \( q \), then the amount of extra reductions drops dramatically. If \( g \) is just slightly larger, then the number of extra reductions decreases again. Since the OpenSSL uses Chinese reminder optimization. This effect is observable for \( p \) and \( q \). The second source of timing leaks is the multi-precision integer multiplication routine. Such integers consist of several words. If the integers consist of an equal amount of words, the Karatsuba multiplication is used, which is executed in \( O(n^{\log_2 3}) \), whereas otherwise the "normal" multiplication is executed, which is executed in \( O(n \cdot m) \), where \( n \) and \( m \) are the lengths of the words. Karatsuba is usually used if \( g \) is just below \( q \), whereas otherwise the "normal" multiplication is, which causes slower computation times [9].

The attack is carried out as follows: Firstly the first half most significant bits of \( q \) are guessed. Then the complete factorization is computed using Coppersmith’s algorithm. Therefore multiple messages are send to a server, where the value of the message is getting closer to \( q \) with each guessed bit. To recover the first 2-3 bits, the decryption times of all combinations of these bits are measured. The resulting plot shows two peaks, one for \( q \) and one for \( p \). Since \( q \) is always smaller than \( p \) in this implementation, the values that bound to the first peak are picked. The values of the next bits are estimated bit by bit. Let \( g \) be a message, where the first bits are the already known bits and all other bits are 0. Let \( g_{hi} \) be a message, where the first bits are the known bits, the first unknown bit is set to 1 and all other bits are 0. Let \( t_1 \) be the decryption time of the Montgomery form of \( g \) and let \( t_2 \) be the decryption of the Montgomery form of \( g_{hi} \). If \( |t_1 - t_2| \) is positive, the extra Montgomery reductions dominated the time difference, otherwise the multi-precision multiplications dominated the time difference. Large intervals indicate, that the bit of \( q \) is 0, whereas small intervals indicate, that the bit of \( q \) is 1. In order to receive more significant results, the authors also took the decryption times of the neighbors of \( g \) into account, so instead of \( t_1 \) they used \( T_g = \sum_{i=0}^{n} DecryptTime(g + i) \) and instead of \( t_2 \) they used \( T_{ghi} = \sum_{i=0}^{n} DecryptTime(g_{hi} + i) \). The amount of neighbors that have to be used depends on the environment. The authors used 400 during their tests and they took seven samples of each decryption time and used the corresponding median [9].

Although the exploitation method is different, the measurement is again done using processor cycles. Nowadays OpenSSL blinding is the default option for RSA encryptions and decryptions. However it is still possible to switch blinding off, and therefore OpenSSL is still a good candidate to study new methods and tools in order to develop methods to find vulnerabilities to timing attacks. This is the reason, why the OpenSSL library will also be used in this thesis.

### 2.4 Morgan and Morgan’s Classification of Potential Timing Leaks

2015 Morgan and Morgan did not show an exploit but instead they studied statistical methods in order to find vulnerabilities to timing attacks in web applications, which usually are very noisy environments [12]. They installed a packet sniffer, to collect outgoing and incoming packets and to expose their TCP timestamps. Their sniffer runs in kernel space because they found, that this approach is less affected by noise compared to an application running in userspace [12].
The measurements were done in pairs, which means that there are always two different input values for which two response times are measured. For instance if there is a field in the web application, which expects a user name, the authors would send a correct user name and an incorrect one and measure both response times. The input which is send first, was randomly chosen for each measurement pair. This measurement method has the advantage, that both samples are affected by changes in the network [12].

In order to classify whether there is a timing difference, they used different tools to find the central tendency of the distribution from the delta between the response times. If this delta is different from zero, then a timing difference exists. The first tool they used to check the central tendency was the midsummary, which is the mean of two selected percentiles, e.g. percentile 40 and 60. The quadsummary is the mean of four different percentiles, whereas the septasummary is the mean of 6 percentiles and the median. They also tried to apply the box test, where one simply computes the low and high percentiles of a distribution. If they do not overlap for two distributions, then these distributions are different from each other. However there are no predefined save parameters regarding which percentiles should be used for the high and the low percentile. The parameters of all those methods were determined by a training on some of the collected data. In addition the authors have also tried to use the Kalman-Filter, which they found very promising, although it had not been implemented into the developed tool, because of the computational complexity [12].

None of the classifiers worked well on deltas which were smaller than 1000 ns. The quadsummary and septasummary classifiers worked well for low noise but were outperformed by the the midsummary classifier if there was more noise. The box test performed worst in most cases [12].

Although this method only targets networks, the work done during this thesis confirms, that detecting the central tendency of the execution times for a sample is not easy to detect. Although the suggested method has not been used for this work, it might still be interesting to try it on a CPU in the future.

2.5 Kaufmann et al. Exploit an Elliptic Curve Diffie Hellman Key Exchange Implementation

The latest timing attack was published in 2017 by Kaufmann et al. It targets the implementation of a 32-bit version of the elliptic Curve25519-donna, which was written by Adam Langley in the programming C [13]. Curve25519 is a carefully designed Elliptic Curve, which is used for Diffie-Hellman key exchanges. The elliptic curve is supposed to be protected against state of the arts timing attacks including cache-timing-attacks. Input-dependent branches and input-dependent array indices are avoided in the design [14]. Nevertheless the compilation of the source code with Microsoft Visual Studio 2015 to a 32-bit binary introduces an unintended conditional branch within the calculation, which results in an exploitable timing leak [13].

In order to optimize the speed of the algorithm, the value of a coordinate from the elliptic curve is represented by polynomials. In order to ensure compatibility for 32 and 64 bit
integers, a design decision was made, which determines, that all 32 most significant bits are 1, if a coefficient is negative and all 32 most significant bits are 0, if the number is positive. If the implementation is compiled with Microsoft Visual Studio 2015 the \textit{jnz} instruction is applied to the most significant bits of an integer. As a result positive coefficients are executed faster than negative coefficients, where a jump occurs [13].

The main idea to exploit this weakness is to send base points to the victim, that cause observable timing variances for a specific key bit, when the victim uses this base point to calculate the shared secret. The attack is executed bit by bit. First of all, the known part of the key is extended, once with a 0 and once with a 1. Both alternatives are then used to execute the code, which calculates the resulting coefficients for the same random base point. Then the number of negative coefficients are counted for both versions. If the difference between the amount of negative coefficients for both versions is at least 8, then the base point is added to a set called $high_0$, if there are more negative coefficients when the key bit is 0, otherwise the base point is added to $high_1$, if there are more negative coefficients if the key bit is 1. This is done until $high_0$ and $high_1$ contain 200 points each. For each base point within the set, the time required by the victim to calculate the shared secret is taken 25,000 times. The mean of the 15 minimum values of each set is taken and compared. If the mean of the 15 minimum values of $high_0$ is larger then the mean of the 15 minimum values of $high_1$, the key bit is 0. Otherwise it is 1. The whole procedure is repeated until the same value for the currently examined key bit is found twice [13].

This exploit shows, that timing attacks are still relevant nowadays. It also proves, that reading the source code is not enough and that it is necessary to test the executable for timing variances. Since this exploit is quite new, it is a good candidate in order to research methods, which are supposed to detect leaks.
Chapter 3

Experiments

The goal of these experiments is to develop a method to detect timing leaks. Therefore a precise measurement method is crucial. The first experiments measure execution times of an encryption method from a cryptography library in Python. Python is a popular programming language, which is often used for web applications and therefore a good target for remote attacks. Popular frameworks are Django, TurboGears and web2py but there are many more. The Python documentation lists more than 50 additional frameworks for web applications [15]. Different measurement methods provided by the Python language itself are compared. Since other programs, which share the computer hardware can cause a delayed execution of the test program, the influence of different scheduling strategies on the test results is examined in an experiment too.

The Python cryptography library that I used for these experiments is written in C. Therefore it is also possible to measure the execution times directly in C, which has the advantage, that this language causes less overhead than an interpreted language like Python and in addition C contains the possibility to use special processor instructions, which are dedicated to high precision benchmarking. Furthermore it is possible to execute these measurements in kernel mode, where only a very small amount of noise occurs. However it is not alway practical to measure execution times, therefore experiments with this measurement method are performed in kernel mode and in user mode.

The measurement method in C is then used to measure execution times and to apply some statistical tools. Besides common tools like the average or mean of a sample, autocorrelation is applied on these results in order to detect pattern in timing differences.

Since large timing differences cannot be expected in a constant time algorithm like AES, the methods studied in the previous experiments are then applied to OpenSSL’s RSA decryption, which contains a known timing leak, if the blinding option is disabled.

3.1 Hardware

All experiments are carried out on a Thinkpad P50s 20FL000DGE, which is running a 64-bit Ubuntu 16.04 using the 4.4.0-67 kernel version. It is equipped with an Intel Core i7-6500U processor and 8 GB RAM. During all experiments, the Notebook was connected to the power network. During the experiments with the language C in kernel space, networking has been disabled, in order to gain more precise results.
3.2 Measurement of Execution Times in Python

The high-level programming language Python offers different approaches to measure the performance of an algorithm. In this chapter, the AES implementation of PyCrypto is used in order to measure execution times. AES is expected to run in constant time, therefore the execution time differences are supposed to very small or zero. Furthermore the effect of attaching test programs to different cores and to different amounts of cores is observed. The most precise results are expected, when the test program is attached to a single core. In addition the effect of using the FIFO scheduler is compared to the effects from the normal scheduler. It is expected, that the FIFO scheduler reduces noise, since other processes have lower chances to gain access to the processor.

3.2.1 Python

The programming language Python\(^1\) was invented by by Guido van Rossum in the early 1990s [16]. The name of Python was chosen because van Rossum was a big fan of "Monty Python’s Flying Circus" at that time. Python is supposed to be an elegant high level object oriented general-purpose language, which is easy to learn and easy to read, since it uses indentation instead of braces to structure the code. It provides a large standard library and there is a wide selection of additional libraries for different purposes. Unlike statically typed languages like C or Java, Python uses dynamic types. The official Python implementation uses an interpreter called CPython, which is written in C and which is available for many platforms. Additionally there are further ongoing projects that are building different interpreters and compilers [17].

Currently there are two Python versions available: Python 2.7 and Python 3.6. Python 2.7 is still supported for backwards compatibility reasons whereas 3.6 is the current version of Python, which is not fully compatible with the version 2.7 and earlier versions. Some features of Python 3.x have been back-ported to Python 2.7. Since Python 3.6 has not been available at the beginning of this project, Python version 3.5 has been used instead. The copyright of the current Python versions is held by the Python Software Foundation. The language is open-source and free to use, even in commercial projects [17].

3.2.2 Python Extensions

Language extensions for Python can be written not only in Python but also in C or C++. Since the PyCrypto library, which is be used during the experiments, is implemented as a C-extension, a short introduction about the general architecture of such a C-extension is given.

A Python extension typically consists of a `.c` file, which includes the header `Python.h` and a `setup.py` file, which is necessary to compile the C-extension. Figure 3.1 shows the basic structure of an extension, where a method from a Python file called `helloPython.py` is implemented as a C-function, which is defined in `hello.c`.

\(^1\)www.python.org.
The method in helloPython.py calls the method from a shared object, which is created, when the source code from the C function is compiled with setup.py. Python variables that need to be passed to a C function are passed as PyObjects, which are defined in the Python.h header. The expected data type is parsed from the PyObject using PyArg_ParseTuple from the header. In order to return values from variables, a function called Py_BuildValue, which is also defined in the Python.h header, saves the value of a specific variable into a PyObject.

In order to build a Python extension, three additional methods or variables have to be added to the C-program. First of all a PyMethodDef table has to be declared, which lists name, function, expected parameters and a documentation string of each function, that should be provided by the extension. Secondly, the module, that is supposed to be build, has to be defined by a structure called PyModuleDef, which requires the name, the module documentation, a per-interpreter state and the name of a table that contains all defined methods as parameters. Thirdly an init function, that initializes a module needs to be implemented.

If a setup.py file is given, the extension is compiled using distutils. Therefore the methods setup and Extension are imported from distutils.core and implemented. The extension method requires a name for the extension and a list of sources. The setup method requires a package name, the version, a description and a list of modules, which should be included into the package.

Further features provided by the Python API for C-extensions can be looked up in the Python documentation on the Python website [17].

### 3.2.3 PyCrypto

PyCrypto[^2] is a toolkit for Python, that contains a collection of implementations of cryptographic algorithms including a random number generator. It aims to provide consistent interfaces for similar types of algorithms. The current version is 2.6.1, which is also the version that has been used for the experiments described in this thesis [18].

[^2]: https://www.dlitz.net/software/pycrypto/
Most cryptographic algorithms are implemented as C extensions. This also applies to
the AES implementation, which will be used for the first experiments. A rough overview
of the implementation is given in figure 3.2. The interface to Python is defined in AES.py.
AES.py imports a class called blockalgo.py, where common methods and variables are de-
defined, which are shared among different block cipher algorithms implemented in this library,
namely AES, Blowfish or DES. AES.py imports a shared object called _AES, which has
been compiled from the corresponding C sources. The C file that implements _AES is
called AES.c. It includes a file called block_template, which contains shared functions and
variables for all block cipher algorithms and most of the declarations, which are required to
compile the extension. AES specific functions and variables are defined in AES.c.

![Figure 3.2: Overview over the AES implementation in PyCrypto](image)

An example of how this library is used is given by figure 3.1. Firstly AES is imported.
The import of Random is only done, because an initialization vector is required, although the
electronic codebook (ECB) mode does not use it. It still has to be defined by the programmer
in order to maintain the common API, that is provided for all types of block ciphers. From
the key, the mode and the initialization vector, an AES object is built, which is then called to
encrypt a message.

```python
from Crypto.Cipher import AES
from Crypto import Random

key = b'Super Secret Key'
message = b'Hello World!!!!!!'

iv = Random.new().read(AES.block_size)
aes_cipher = AES.new(key, AES.MODE_ECB, iv)
encrypted_message = aes_cipher.encrypt()
```

The initialization process of an AES object is shown in figure 3.3. The AES object is defined
by a class called AESCipher, which is implemented in AES.py. The class AESCipher has
a superclass called BlockAlgo, that is defined in blockalgo.py and which is included in AES.py. The required blocksize is provided by the C-extension and is defined in AES.c. To create a new corresponding AES-object in C, the function ALGnew from block_template.c is called, which then calls the function newAlgObject, that creates a structure called AlgObject. ALGnew also calls the variable block_size and the function block_init, which are defined in AES.c. block_init receives pointers to the ALGobject structure and calls rijndaelKeySetupEnc and rijndaelKeySetupDec. The first function expands the key for encryption and the second function expands the key for the decryption. The results affect the variables in the ALGobject structure. The returned block_size is saved to the structure by ALGnew. The resulting ALGobject structure corresponds to the Python object, which is defined by the classes AESCipher and BlockCipher.

The encryption process is shown in figure 3.4. To encrypt a message, the method encrypt from the AESCipher object is called. Since all methods are defined in the superclass BlockAlgo the encrypt method of BlockAlgo is called, which then calls the ALGEncrypt function in block_template.c. ALGEncrypt chunks the message into blocks according to the variable BLOCKSIZE defined in AES.c and parses the required values from the PyObject. For each of these message blocks the function block_encrypt is called, which is also defined in AES.c. The messages and the buffer for the resulting encrypted messages are handed over as pointers. block_encrypt calls the function rijndaelEncrypt, which encrypts the blocks and saves the encrypted blocks into the spaces that are indicated by the pointers. The encrypted message, represented as a PyObject in C, corresponds to a bytestring in Python. This is the returned result, which is returned by Python.
Figure 3.4: Encrypt Blocks
3.2.4 Measurement Methods in Python

The time module in Python 3.5 offers five different ways to measure the performance of a program [19]. The current implementation of a time measurement methods on a system and its resolution can be viewed using the command `time.get_clock_info('name of a Python clock')`. The resolutions of all different time measurement methods are given in table 3.1. In Ubuntu 16.04, all Python time measurement methods are implemented using a clock of the GNU library glibc version 2.23 [20]. Further information about these clocks can be retrieved by typing `man clock_gettime` into the terminal. The elapsed time is always calculated from the delta between two clock calls. The available Python functions are the following:

- `time.time()` calls `clock_gettime(CLOCK_REALTIME)` in Linux systems and returns a float, which represents the current wallclock time in seconds. It can be set by an administrator and it also adjusts frequently in an incremental manner.

- `time.monotonic()` calls `clock_gettime(CLOCK_MONOTONIC)`. It represents the time that has passed since the Epoch, which is January 1, 1970, 00:00:00 (UTC) on most Unix systems. The Epoch which is used can be checked in Python using the command `time.gmtime(0)`. Just like the previous clock, this clock frequently adjusts in an incremental manner.

- `time.clock()` uses the clock() function of glibc in Linux, which is implemented on top of the `CLOCK_PROCESS_CPU_ID` clock since glibc version 2.18. It returns an estimation of the processor time, which has been used by the current program in seconds. The time wraps around at some point. `time.clock()` is deprecated since the Python version 3.3 and it is recommended to use one of the clocks, that are described next.

- `time.perf_counter()` is a performance counter with the highest available resolution on the system and is intended to be used for short benchmarking durations. Its behavior is supposed to be well defined on all platforms. On the system, which is used for this thesis, it is implemented using `CLOCK_MONOTONIC`.

- `time.process_time()` represents the sum of kernel and user space CPU time. Its implementation uses `CLOCK_PROCESS_CPUTIME_ID` from the glibc library, which provides a high-resolution per-process timer from the CPU. All threads of a process are taken into account.

- `timeit.timeit()` is a module to measure execution times of statements. It measures the time using `time.perf_counter()` in the default case, but it can also use any other clock instead. During the measurements, the garbage collection is switched off. However this can also been archived manually with `import gc` and `gc.enable()` or `gc.disable()`. Furthermore `timeit.timeit()` is supposed to optimize the way how statements or callables are called and it can make multiple measurements to improve the precision of the results.

There are also many other libraries to benchmark Python codes, but to my knowledge, they are all implemented using one of the five previously described clocks.
Since there is more to test than a simple statement, when measuring the encryption time of a block, \texttt{timeit.timeit()} does not seem to be suitable for this case, although it might be possible to inject additional methods and definitions using the \texttt{globals} variable. \texttt{time.clock()} has a lowest resolution of all clocks, therefore the other clocks may be more suitable. \texttt{time.time()} and \texttt{time.monotonic()} have an equally high resolution. Both clocks have the drawback, that only the delta between the function calls can be measured. The method that is tested might not be running all the time between two timestamps, because there are other processes from the other programs, which are executed in parallel and compete to use a processor core. \texttt{time.time()} has the drawback, that it is possible to set the time, therefore it has to be ensured, that no administrator changes the current wall clock time during the tests. As a result, \texttt{time.process_time()} seems to most suitable, because the results should be less biased by the execution time of other competing processes and the resolution is very high. The next experiment shows, that this assumption is wrong.

### 3.3 Measurement in Python with Different Schedulers on Different Cores

To determine, which measurement method is the most suitable in Python, different schedulers and different amounts of noise on different cores and different amounts of cores are combined with the previously described five clocks. The aim of this experiment is to find out how scheduler, exclusive access to core and different measurement methods influence the results.

#### 3.3.1 Setup

The different setups tested in this experiment are given in table 3.2. Some of the setups use the FIFO scheduler and a process priority of 99, which is the highest priority available for this scheduler. The FIFO scheduler is intended for real-time applications and it does not use any time slicing. Instead the thread with the highest priority is executed first. If there are several threads with the same priority, the first process in the list is executed first. However there are some instructions, which make it possible to place a thread with the same priority ahead of the other threads. If a thread with a higher priority becomes ready to be executed, threads with a lower priority are interrupted. In order to keep the system usable, the default settings only allow the FIFO scheduler to use 95% of the time. The rest of the time is given to the other threads in order to keep the system functional [20]. However this only applies to those threads, which are not scheduled using FIFO. This is the reason, why programs like

<table>
<thead>
<tr>
<th>Python Clock</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{time.time()}</td>
<td>$1e^{-9}$</td>
</tr>
<tr>
<td>\texttt{time.monotonic()}</td>
<td>$1e^{-9}$</td>
</tr>
<tr>
<td>\texttt{time.clock()}</td>
<td>$1e^{-6}$</td>
</tr>
<tr>
<td>\texttt{time.perf_counter()}</td>
<td>$1e^{-9}$</td>
</tr>
<tr>
<td>\texttt{process_time()}</td>
<td>$1e^{-9}$</td>
</tr>
</tbody>
</table>

Table 3.1: Clock Resolutions
the network manager crashed during the test. As the command `ps -eo pid,rtprio,cmd` reveals, a process connected to the wireless LAN driver uses the FIFO scheduler and has a priority of 50. Therefore it was never executed during the experiment.

All the tests, which are not run in FIFO mode, simply use the normal scheduling mode of the completely fair scheduler (CFS) on Linux. The CFS aims to share the CPU time equally among all threads. Therefore every single thread has field, which indicates the time, that the process already ran. The scheduler allows each thread to run for a while. When the thread schedules or when the scheduler tick happens, the next thread is chosen from a red-black-tree, which stores all threads according to their timestamp. If a thread should receive a higher or lower priority, a nice value can be set. The amount of extra time, which the thread gains by changing the nice value depends on the nice values of all other threads. Furthermore there are two more scheduling strategies, which are also controlled by the CFS. Threads, which are using the batch strategy are called less frequently but receive more time, when they are scheduled. Threads, which use the idle strategy are treated with a very low priority. The default strategy is however the normal mode, where threads get a short execution time frequently to provide a highly responsive system [21].

The Thinkpad, which was used during the experiments is equipped with two hardware cores but four virtual cores. The virtual cores 0 and 2 share one physical core and the virtual cores 1 and 3 share one physical core. The cores, which are used by the test-program are chosen in a way, that each test is forced run on a different core. Some of the cores in the table are highlighted. These are the cores, where noise can be expected, since the operating system and other programs running in the background are using them. Two virtual cores have been isolated using the boot parameter `isolcpu=1,3` in grub2, therefore one can expect less noise here. As a result, a whole physical core remains unused, as long as no other process is manually attached to one of its virtual cores. The test processes are attached to the cores by using the command `taskset`. During the experiment, all cores are set to performance mode.

For each setup and each clock, the execution time of the encryption process of the message `Attack at dawn!!` and key `Sixteen byte key` was measured 1000200 times. The first 200 results were removed to ensure, that the processor and the cache have been warmed up.

To get an idea about the general performance of the clocks, a second experiment is carried out, where each clock measures the execution time of each other clock. Therefore the total execution time of 1000 iterations of each clock is measured using `timeit()` and the corresponding other clocks. This test is repeated 1 000 000 times for each clock. Afterwards the minimal value of each test is taken and divided by the number of iterations.

### 3.3.2 Results

Regarding the encryption times, the results for all tests show only small variances for each timer, whereas the measured execution times of the different timers differ a lot more. Table 3.3 shows the results of each timer on a single core with noise. `time()` and `clock()` are the only timers, that always deliver the same shortest execution time during each test and they are also the only measurement methods, which detect this minimal result not only once or twice but several times. `time()` returns its minimal result between 43292 and 302630 times among the different setups and furthermore it returns the lowest execution time of
### 3.3. Measurement in Python with Different Schedulers on Different Cores

<table>
<thead>
<tr>
<th>Test Nr</th>
<th>Scheduler</th>
<th>Taskset</th>
<th>Cores offered</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FIFO + priority 99</td>
<td>yes</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>FIFO + priority 99</td>
<td>yes</td>
<td>1,2</td>
</tr>
<tr>
<td>3</td>
<td>FIFO + priority 99</td>
<td>yes</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>FIFO + priority 99</td>
<td>no</td>
<td>0,2</td>
</tr>
<tr>
<td>5</td>
<td>FIFO + priority 99</td>
<td>yes</td>
<td>1,3</td>
</tr>
<tr>
<td>6</td>
<td>standard</td>
<td>yes</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>standard</td>
<td>yes</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>standard</td>
<td>yes</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>standard</td>
<td>no</td>
<td>0,2</td>
</tr>
<tr>
<td>10</td>
<td>standard</td>
<td>yes</td>
<td>1,3</td>
</tr>
</tbody>
</table>

Table 3.2: Test Scenarios for Time Measurements in Python

<table>
<thead>
<tr>
<th>Clock</th>
<th>Fastest Encryption Time</th>
<th>Average Encryption Time and Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>time.time()</td>
<td>$7.15e-07$ $(281537x)$</td>
<td>$9.53e-07 \pm 6.0461e-07$</td>
</tr>
<tr>
<td>time.monotonic()</td>
<td>$8.26e-07$ $(1x)$</td>
<td>$9.59e-07 \pm 8.2326e-07$</td>
</tr>
<tr>
<td>time.clock()</td>
<td>$1.00e-06$ $(69742x)$</td>
<td>$1.18e-06 \pm 5.9901e-07$</td>
</tr>
<tr>
<td>time.perf_counter()</td>
<td>$8.53e-07$ $(1x)$</td>
<td>$9.98e-07 \pm 8.3447e-07$</td>
</tr>
<tr>
<td>time.process_time()</td>
<td>$1.03e-06$ $(2x)$</td>
<td>$1.17e-06 \pm 5.0701e-07$</td>
</tr>
</tbody>
</table>

Table 3.3: Average Encryption Times for on a Single Core with noise

At first sight, the FIFO scheduler seems to have only a small influence on the amount of noise in the measurements. The same applies to the number of cores, which are offered and the amount of noise. The mean execution time and the standard deviation seems to be slightly lower when the test-program uses one of the cores which contains less noise and also if two cores are used, but the difference is too small to be significant. In general the standard deviations are quite high, because they are massively affected by outliers with a very slow execution time.

Table 3.4 shows the small differences among the different setups for `time()`. The fastest execution times are always the same and detected frequently. Nevertheless the FIFO scheduler, and executions on single cores, which are not used by any other processes or the operation system, increase the amount of fastest executions. This effect is visualized in figure 3.5.

Table 3.5 shows the results of the time measurements of the different clock executions. The fastest execution time is measured for the `monotonic` clock, no matter which timer has all timers. The result of `clock()` also very stable, but the larger standard deviation indicates, that this measurement method may not be sensitive enough to detect timing differences.

`perf_counter()` and `monotonic()` deliver quite similar results. Their shortest detected execution times are however more than 10 ns longer than those of `time()`. `process_time()` measures longer times than all the other clocks and according to the standard deviation it is less precise, than expected.
CHAPTER 3. EXPERIMENTS

<table>
<thead>
<tr>
<th>Test</th>
<th>Fastest Encryption Time</th>
<th>Average Encryption Time and Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO, noise, 1 core</td>
<td>7.15e − 07 (302630x)</td>
<td>9.43e − 07 ± 3.9047e − 07</td>
</tr>
<tr>
<td>FIFO, noise, 2 cores</td>
<td>7.15e − 07 (127457x)</td>
<td>9.91e − 07 ± 4.2820e − 07</td>
</tr>
<tr>
<td>FIFO, noise 2 phys. cores</td>
<td>7.15e − 07 (84306x)</td>
<td>1.01e − 06 ± 4.3797e − 07</td>
</tr>
<tr>
<td>FIFO, no noise, 1 core</td>
<td>7.15e − 07 (235063x)</td>
<td>9.20e − 07 ± 3.1092e − 07</td>
</tr>
<tr>
<td>FIFO, no noise, 2 cores</td>
<td>7.15e − 07 (164035x)</td>
<td>9.37e − 07 ± 2.9628e − 07</td>
</tr>
<tr>
<td>noise, 1 core</td>
<td>7.15e − 07 (281537x)</td>
<td>9.53e − 07 ± 6.0461e − 07</td>
</tr>
<tr>
<td>noise, 2 cores</td>
<td>7.15e − 07 (274005x)</td>
<td>9.55e − 07 ± 8.2729e − 07</td>
</tr>
<tr>
<td>noise, 2 phys. cores</td>
<td>7.15e − 07 (43292x)</td>
<td>9.70e − 07 ± 3.0546e − 07</td>
</tr>
<tr>
<td>no noise, 1 core</td>
<td>7.15e − 07 (215324x)</td>
<td>9.25e − 07 ± 3.1807e − 07</td>
</tr>
<tr>
<td>no noise, 2 cores</td>
<td>7.15e − 07 (261497x)</td>
<td>9.14e − 07 ± 3.2714e − 07</td>
</tr>
</tbody>
</table>

Table 3.4: Average Encryption Times of \texttt{time()}
been used. However the equally implemented perf_counter is slower. time delivers the second fastest execution time for all measurements. clock and process_time perform a lot slower than the other clocks. This test was run several times to ensure, that the results do not represent a bad sample. Sometimes time was faster than monotonic and sometimes monotonic was faster than time. The result is not very stable.

### 3.3.3 Discussion and Conclusion

The good results for time() are surprising, since the Python documentation recommends to use perf_counter() or process_time instead. Since time() delivers the best results for small amounts of time and since there is no administrator, who could reset the time in the middle of a test, this method will be used in further experiments, because it seems to be the most reliable way to estimate small timings. Figure 3.6 shows, that this method mostly fluctuates between two values.

![Signal of time.time()](image3.png)

Since perf_counter() is based on monotonic() it is no surprise, that their results are quite similar. It is however surprising, that process_time delivers such bad results. Since this method is supposed to only take the time into account, which is spend in kernel and user mode by a specific process, I would have expected, that its resulting time measurements are less biased than those from perf_counter().

![Encryption Number](image4.png)
The higher execution times that appear, when `process_time` or `clock` are used, may be caused by overhead, which is clearly visible, when the execution time of the clock itself is measured.

In contrast to the choice of a time measurement method, the scheduling policy, process priority and attachments of processes to specific cores seem to be less crucial. With enough measurements it is always possible to determine the correct shortest execution time, but it might take a higher amount of samples. Using two different physical cores seems to be a bad idea in general, but if two emulated cores on the same physical core are used, the impact on the results is lower.

### 3.4 Execution Times of Different Keys and Different Blocks in Python

Having determined the best time measurement method in Python, the next step is to check if it is possible to detect timing differences in the AES implementation of PyCrypto. In a first experiment, a simple sixteen byte block is encrypted by different keys. If there are any key-dependent execution time variances, then they might be detectable. The second experiment, different sixteen byte blocks are encrypted with several keys. Therefore if there are timing differences, which are caused by different messages, this might be detectable. Furthermore if there are timing differences, which appear only for specific key and message combinations, these effects may be detectable too.

#### 3.4.1 Setup

Within the first experiment the sixteen byte block, which is encrypted is the block "Attack at dawn!!". This block is encrypted with 1000 different keys. For each key, the measurement is repeated 1000 times. Since noise always increases the true execution time, the shortest encryption time and its quantity is determined for each key. The first key is the previously used 16 byte key "Sixteen byte key", the second key consists of zeros and the third key consists of ones. The remaining 997 keys consist of sixteen randomly generated bytes.

In the second experiment, the first 200 keys from the previous experiment are used to encrypt 1000 different messages each. The measurements are repeated 500 times for each key and each block. The reason, why only so few keys are used and why only so few iterations have been executed in this test, is because a lot of noise was observed, which seems to be introduced by some list methods, when they become very long. Therefore larger samples lead to less stable results. The first message is again the previously used "Attack at dawn!!". The second message only consists of zeros and a third message only consists of just ones. All other messages consist of sixteen randomly generated bytes.

Both tests are attached to a single virtual core using `taskset`, which is not occupied by the operating system and where the whole physical core is not shared with the operating system either. Because additional noise seemed to be introduced by an occupied RAM, the computer was restarted before the second experiment and networking has been disabled.
3.4.2 Results

In the first experiment, all keys have the same shortest encryption time, which is $7.152557373046875e - 07$. The occurrence of the shortest encryption time varies from 555 to 814.

The same applies to the second experiment. The shortest encryption time is also $7.152557373046875e - 07$ no matter which key or message is used and the occurrence of the shortest execution time varies from 250 to 438.

3.4.3 Discussion and Conclusion

In contrast to Bernstein, I am only trying to detect leaks and therefore it is not necessary to examine, which bytes determine higher or lower execution times. However if setting a specific byte to a specific value should frequently lead to higher execution times for those messages containing that byte, and therefore this should lead to higher execution times for many of those messages. Therefore such an effect should become visible in the previously performed experiment as well. If the sample size of the randomly generated messages is large enough, there should always appear some messages containing at least one byte, that leads to higher execution times compared to messages, that do not have such a byte. In order to catch these cases, the sample size needs to be large enough. The same applies to key-dependent execution time variances.

The results of the two experiments show, that the AES implementation may really perform in constant time. However there are two potential error sources. First of all it is possible, that the measurement method is not precise enough to detect small differences. Furthermore it is possible, that the amount of randomly generated keys and messages is too small, so that combinations of bytes, which are increasing the execution times, have not appeared. Therefore it is still possible, that a key-message-combination exists, which introduces a timing leak.

3.5 Measurement in C

Within the last experiments, the execution times were measured using functions from Python. Since the PyCrypto library is implemented in C, its performance should also be measurable in C. The advantage of C is, that C is not an interpreted language, so there is no overhead from the interpreter.

One approach to measure the performance of a piece of software on an Intel processor using the C programming language has been described in the Intel Whitepaper "How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures" by Gabriele Paoloni [22]. This method has two advantages. Besides the advantages from the C programming language, it uses some Intel specific low-level processor instructions dedicated to high precision benchmarking. Within the next subsection, the suggested method is described thoroughly. Afterwards the previous experiment is repeated using this new method.
3.5.1 The Kernel Method suggested by Intel

According to the Intel whitepaper, the execution times are should be measured in kernel mode to guarantee the exclusive ownership of the processor. The proposed programming language C furthermore offers the possibility to disable CPU preemption and hard interrupts before the test. These are then re-enabled again after the test. This is supposed to avoid any possible interruption during the measurements. A combination of the Intel processor specific instructions CPUID, RDTSC and RDTSCP is used to serialize the instruction stream and to measure the execution time precisely. The CPUID instruction is used as a barrier, that guarantees, that all preceding instructions have been executed and that no subsequent instruction is executed before the CPUID instruction. This is necessary because current processors support out of order execution. Therefore instructions are not necessarily executed in the order in which they are written. Instead they may be reordered to increase the throughput of the processor. The RDTSC instruction is used to read a timestamp from a special register. Since Pentium 4, the time which is measured by the time stamp counter, increases monotonically and independently from the current speed of the processor cores [23]. The RDTSCP instruction does the same as the RDTSC instruction, but it additionally reads the cpu id within the same step. Furthermore it guarantees, that all previous instructions have been executed. It does however not guarantee, that the subsequent instructions are not executed. This is the reason, why CPUID is called afterwards again. The results of RDTSC and RDTSCP are stored in the registers eax and edx. RDTSC also uses the ecx register and CPUID additionally uses the ecx and the ebx register. Since the first 32 bits of the 64 registers are cleared, rax, rbx, rcx and rdx have to be declared as clobbered registers, to ensure, that the compiler knows about the modification of these registers. The source code to measure the time is shown in listing 3.2.

Listing 3.2: Performance measurement according to Intel [22]

```c
preempt_disable(); // disable preemption on CPU
raw_local_irq_save(flags); // disable hard interrupts on CPU
asm volatile(
  "CPUID\n"
  "RDTSC\n"
  "mov %edx, %0\n"
  "mov %eax, %1\n": "=r" (cycles_high), "=r" (cycles_low)::"%rax", "%rbx", "%rcx", "%rdx";

/* call the function to measure here */
asm volatile(
  "RDTSCP\n"
  "mov %edx, %0\n"
  "mov %eax, %1\n": "=r" (cycles_high1), "=r" (cycles_low1)::"%rax", "%rbx", "%rcx", "%rdx";

raw_local_irq_restore(flags); // enable interrupts
preempt_enable(); // enable preemption

start = (((unsigned long)cycles_high << 32) | cycles_low);
end = (((unsigned long)cycles_high1 << 32) | cycles_low1);
time = end - start;
```
3.5. MEASUREMENT IN C

3.5.2 Kernel Programming

The following includes are required by the test routine, that determines the execution times of the AES encryption:

Listing 3.3: Required Includes of the Test Routine

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/hardirq.h>
#include <linux/preempt.h>
#include <linux/slab.h>
```

The first include is mandatory for every kernel module. The second include is required to print a log message into the ring buffer with a specific log level [24]. The `hardirq` and `preempt` macros are required to disable preemption and hard interrupts. The last macro is required to allocate and free memory on the heap.

Instead of a main function two functions with exactly the names `int init_module(void)` and `int cleanup_module(void)` have to be implemented. The first function calls the test routine, and the second function simply returns. An alternative is to include another macro called `linux/init.h` and call the defined functions using `module_init('function')` and `module_exit('function')` [25].

Another property of kernel modules is, that the ISO C90 standard is enforced, and therefore it is not possible to mix declarations with code. Furthermore file operations are not as straightforward as usually in C. An easy way to get the keys and blocks that are required into the program is to write a program, that writes a `.c` file, which fills the corresponding buffers with the required data from a file and include this `.c` file. There are more elegant ways to deal with this issue, but for the next experiments, this method is good enough.

Since the stack of the kernel is quite small, space for the test data has to be allocated in the heap. Instead of `malloc` and `free`, the kernel versions `kmalloc` and `kfree` have to be used, where `kmalloc` expects one parameter more than `malloc`, which makes it possible to allocate memory with different properties. In this case just the normal memory allocation procedure is required, so `GFP_KERNEL` is chosen [26]. Furthermore instead of `printf`, `printk` has to be used. It also requires an additional parameter, because kernel messages have different log levels, which determine, if the messages are shown in the console and in which log file they might be found. All `printk` messages are saved in a ringbuffer, which is overwritten, when it is full. Therefore it was necessary to increase the ringbuffer by adding the boot parameter `log_buf_len=2G` to the line starting with `GRUB_CMDLINE_LINUX_DEFAULT` in the file `/etc/default/grub`. There is another process, which fetches the data from the ringbuffer and logs the messages in `/var/log/syslog` [26].

The easiest way to compile a kernel module is to use a makefile 3.4. After compiling, the module can be loaded with the command `insmod AES-timings.ko` and removed with the command `rmmod AES_timings` [24].

Listing 3.4: Makefile for Kernel C
CHAPTER 3. EXPERIMENTS

1. `obj-m +=AES-timings.o`

3. `all:
   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules`

5. `clean:
   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean`

3.6 Measurement with One Key and One Block

The target of this first experiment is to gain more knowledge about the benchmarking method described above. Therefore the message "Attack at dawn!!" and the key "Sixteen byte key" are used for 250,000 encryptions. The measurement results are stored in an array and after all measurements have been performed, they are printed into the ringbuffer using `printk`.

3.6.1 Setup

The measurement approach needs to be implemented as described above. Since AES encryption from the PyCrypto library from the previous experiment is a Python extension written in C, its source code is used for the next experiments. To measure the encryption execution times, the file AES.c from the PyCrypto extension has to be included, which provides functions to encrypt a single 128-bit message using a 128-bit key. Two functions which are defined in that class have to be called in order to perform the experiment. The first function expands the key:

```c
rijndaelKeySetupEnc(k, key, keyBits);
```

`key` is an array, where the key current key is saved and `keyBits` is an integer value indicating the key length, which is 128 in this test. `k` is another array of length 4*(rounds+1)+1, where the expanded key is stored. The rijndael algorithm requires 10 rounds for 128 bit keys.

The whitepaper recommends, that the function, whose execution time should be measured is inlined. Therefore the encryption function, which has to be called next is inlined:

```c
rijndaelEncrypt(k, rounds, message, encryptedMessage);
```

It takes the expanded key `k`, the number of rounds, which is again equal to ten, a 128 bit long message `message` and an array called `encryptedMessage`, where the encrypted message should be stored. In order to ensure, that the function is not optimized away, it is important to do something with the result after the measurement. Alternatively it is possible to declare the result variable to be `volatile`, which forces the compiler to update the value in every loop. The latter strategy has been chosen for this test. Furthermore all user space includes have to be removed from the source file, otherwise the compiler rejects the file.

The first attempt to measure the execution times resulted in a very high standard deviation and it was visible, that the system took a long time to stabilize. Therefore the setup was modified. The first setup used two virtual cores, which were located on a single physical core. The second setup used two virtual cores, where each of the cores was located on a different physical core. The other virtual cores were disabled. The third setup used only a single virtual core, while all other cores are disabled. For each setup, the key Sixteen byte key was used to encrypt the message Attack at dawn!! 250 000 times.
3.6. MEASUREMENT WITH ONE KEY AND ONE BLOCK

3.6.2 Results

Even though all interrupts and preemptions are supposed to be switched off, the traces of all 250000 measurements show some very high outliers for all setups. The setup which uses two virtual cores on a single physical core is affected most. The resulting trace is represented by figure 3.7.

![Figure 3.7: Trace of 250 000 measured Encryptions](image)

A closer look at the first and last 250 encryptions in figure 3.8 shows, that the first execution times vary a lot. Furthermore it is visible, that the first measurement result is quite high compared to the other measurement results. In contrast the last 250 measurement vary a lot less.
The setup using two physical cores, which is displayed in figure 3.9 needs much less time to warm-up. Only the first encryption takes a very long time. Afterwards the measurement results stabilize.

Figure 3.10 is a zoomed version of the previous figure 3.9 presenting only the first 250 execution times.

The trace of the third setup, which uses only a single core looks similar to the second setup.
3.6. MEASUREMENT WITH ONE KEY AND ONE BLOCK

Table 3.6 summarizes the results for all setups. The fastest execution time is always 220 cycles. The average encryption time of the first setup is a lot higher than the average encryption time in the other setups. The medians of all setups are nearly equal. However the medians tend to vary a little bit, if the tests are executed repeatedly. It usually ends up between 228 and 230 cycles. The slowest execution time in the third setup is a lot slower than the slowest execution time of the first setup. However it is a matter of luck, how much noise appears in the system and how much it can delay the encryption time during the experiment. If run repeatedly, the fastest encryption time also varies slightly. Nevertheless, the slowest encryption time of the first setup has always been way higher than the slowest encryption time in the last setup. The same applies to the standard deviation, which is heavily influenced by the outliers too. The standard deviation of the first setup is a lot higher than the standard deviation of the last setup.

In contrast to the previously performed tests with Python, the fastest encryption times are rarely appearing outliers. The encryption time of 220 cycles appears 15 times in setup one, three times in setup two, and 21 times in setup three. Sometimes the fastest execution time is 222 cycles instead of 220. The distribution of the encryption times is visualized in figure 3.11. The first subplot shows the overall distribution of the first setup for measured encryption execution times, that are lower than 500. Most executions take between 200 and 250 cycles. These peaks also appear in the plots of the other two setups. The next three subplots show a zoomed version of the peak for each setup. The most often measured encryption time is 228 cycles, followed by 230 cycles. However this value is also not too stable if the test is repeated, the highest peak sometimes appears at 228 cycles instead of 230.

<table>
<thead>
<tr>
<th>Execution Times</th>
<th>1 Core 2 Threads</th>
<th>2 Cores 2 Threads</th>
<th>1 Core 1 Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fastest</td>
<td>220</td>
<td>220</td>
<td>220</td>
</tr>
<tr>
<td>Average</td>
<td>241.30</td>
<td>230.55</td>
<td>228.98</td>
</tr>
<tr>
<td>Median</td>
<td>230.0</td>
<td>230.0</td>
<td>228.0</td>
</tr>
<tr>
<td>Slowest</td>
<td>36382</td>
<td>7730</td>
<td>428</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>167.3948</td>
<td>41.8602</td>
<td>2.3263</td>
</tr>
</tbody>
</table>

Table 3.6: Statistics of Different Setups
Figure 3.11: Distribution of Encryption Times for all Setups

3.6.3 Discussion and Conclusion

Since the level 1 cache, execution units, instruction fetch and decoding units are shared only between virtual cores, which run on the same physical core [27], it is not surprising, that the standard deviation of the first setup is so much higher than standard deviation of the other two setups. Even though the program has exclusive access to a processor core, it might still have to wait for resources, that are also used by the other virtual core.

The explanation for the high standard deviation in the second setup is given by the System Monitor. The workload on both physical cores increases during the tests, which means, that both cores are used for the test. Since physical cores can only share information using the level 2 cache, which is slower than the level 1 cache, the measured execution time becomes slower, when the cores have to exchange their information.

Therefore all future measurements should be done using the third setup, which only uses a single core. Furthermore the first measurement value should definitely always be ignored. It might also make sense to ignore more than just the first value just to ensure, that the system has stabilized.

Although the measurement method within the kernel is affected by less noise, there still
3.7 AUTOCORRELATION

seem to be some interruptions. Furthermore it also turned out, that a combination of CPUID, RDTSC(P) and the rest of the source code has to be done carefully. For instance if the measurement results are not collected in an array but printed out immediately using printk, a weird stair-like patterns appears, if the trace is plotted. An example is given in figure ??.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig3_12.png}
\caption{Printing measurement results out immediately with printk}
\end{figure}

3.7 Autocorrelation

Autocorrelation is a mathematical method used in signal processing to find patterns in a noisy signal. Therefore the signal is compared with all delayed versions of itself, giving rise to a new signal. The x-axis of this new signal represents lags instead of time. A lag is the delta between the original signal and the delayed signal. For instance let \((v_0, v_1, ..., v_n)\) be a signal, which has been received at the points of time \((t_0, t_1, ..., t_n)\), then values \((v_0, v_1, ..., v_n-h)\) are compared to \((v_h, ..., v_n)\) at lag \(h\) [28].

Depending on the author and implementation, slightly different autocorrelation formulas are given. Here the Python method \texttt{acf} from \texttt{statsmodels.tsa.stattools} is used. It returns all autocorrelations of a time series until a specified lag \(k\). The resulting autocorrelations are values between \(-1\) and \(1\). \(1\) indicates a strong correlation, \(-1\) indicates a strong anti-correlation and \(0\) indicates no correlation at all [29].

The plots in Figure 3.13, 3.14, 3.15 and 3.16, show some example autocorrelation plots for some given signals. The first plot shows a simple sine-shaped wave. In order to calculate the autocorrelation, the corresponding y-values for \(x = 1, 2, ..., 500\) are taken as input values. Whenever the lag equals a multiple of the period of the sine-shaped wave, the autocorrelation plot shows very high positive values. Whenever a lag is equal to a multiple of half of the period of the sine wave, the autocorrelation shows a very high negative peak, since it the delayed signal is then completely opposite to the original signal. The second signal in the first figure shows random noise. Since it is not possible to detect any pattern using autocorrelation, all autocorrelations are close to zero.
The next figure shows a signal, that only consists of positive signals. High peaks in the autocorrelation plot appear again, whenever the signal is detected. But since there is no negative signal, that is opposed to the positive signals, there are no big negative peaks in the plot. The second plot of the second figure shows the what happens, when noise is added to the signal. Even though the peaks in the autocorrelation plot are lower, it is still possible to detect the signal.

The autocorrelation method has already been used for security testing in the past. It has been used to ensure, that the resulting cipher-texts from cryptographic algorithms, which additionally make use of a pseudo-random number generator do not show any reoccurring patterns, that might leak information about the hidden secret [30]. Furthermore it has been used to reason about the quality of encryption of pictures, which often contain high autocorrelations that have to be hidden [31].
3.7. AUTOCORRELATION

3.7.1 Implementation in Python

The Python method used to evaluate the next experiments is acf from statsmodels.tsa.stattools. A simplified version of the source code is given by Listing 3.5.

acf takes a time series x, which contains the values \((x_0, x_1, ..., x_{n-1})\) as input and calls the method acovf, which is defined in the same source file. The method acovf receives the series of measured timings from acf and subtracts the mean of the time series from each value of the time series. Afterwards, a variable called d is defined. There are two options to calculate d: One is called ”biased” which is the standard option for this formula and the other option is called ”unbiased”. If the unbiased option is chosen, then d is represented by the array \([1, 2, ..., n – 1, n, n – 1, ..., 2, 1]\), where n is the length of the time series. Otherwise d is an array of the same length \(2 \cdot n – 1\), which contains the value n on each position. Afterwards the method correlate from numpy is called.

Numpy’s correlate is implemented as a C-extension. The C implementation is provided by a shared object but the source code is not provided by the numpy package. The documentation within the source code defines the formula which is used to be

\[ a[n+k] = \sum n a[n+k] \cdot \text{conj}(v[n]). \]

Since this application only uses real numbers, the conjugation can be ignored. \(c_{\text{av}}[k]\) can also be written as \(c_{\text{av}}(h)\). Then the input value for \(a[n+k]\) corresponds to \(x_{i+h} – \mu\) and \(v[1]\) corresponds to \(x_i\) and

\[ c_{\text{xx}}(h) = \sum_{i=0}^{n-1-h} x_{i+h} \cdot x_i. \]

The full option, which is set by acovf, is responsible for returning an array containing the values for \([c_{\text{av}}[n-1], c_{\text{av}}[n-2], ..., c_{\text{av}}[1], c_{\text{av}}[0], c_{\text{av}}[1], ..., c_{\text{av}}[n-2], c_{\text{av}}[n-1]]\). Since a and v are the same time series, this set can also be expressed by \((c_{\text{xx}}(n – 1), c_{\text{xx}}(n – 2), ..., c_{\text{xx}}(1), c_{\text{xx}}(0), c_{\text{xx}}(1), ..., c_{\text{xx}}(n – 2), c_{\text{xx}}(n – 1)).\)

acovf then divides each value of the received array by a value of d, and then it removes the first \(n\) values from the list, which afterwards contains the values \((c_{\text{xx}}(0)/n, c_{\text{xx}}(1)/n, ..., c_{\text{xx}}(n – 2)/n, c_{\text{xx}}(n – 1)/n)\) in the biased case and \((c_{\text{xx}}(0)/n, c_{\text{xx}}(1)/(n – 1), ..., c_{\text{xx}}(n – 2)/2, c_{\text{xx}}(n – 1)/1)\) in the unbiased case. This list is then returned to acf.

acf divides each element of the received list by the first item of that list. The first element of the list is \(c_{\text{xx}}(0) = \sum_{i=0}^{n-1-h} (x_{i+h} – \mu) \cdot (x_i – \mu)\) equals \(\sigma^2\). Afterwards the first \(h + 1\) elements of that list are returned, where \(h\) is the requested number of lags.

```
Listing 3.5: Source code of acf
1 def acf(x, unbiased=False, nlags=40, qstat=False, fft=False, alpha=None):
2     ... avf = acovf(x, unbiased=unbiased, demean=True, fft=fft)
3     acf = avf[:nlags + 1] / avf[0]
4     ...
5     return acf

8 def acovf(x, unbiased=False, demean=True, fft=False):
9     ...
10    if demean:
11        xo = x – x.mean()  
12     ...
13     n = len(x)
```
... unbiased:
    xi = np.arange(1, n + 1)
    d = np.hstack((xi, xi[:-1][:1]))
else:
    d = n * np.ones(2 * n - 1)
if fft:
...
else:
    acov = (np.correlate(xo, xo, 'full') / d)[n - 1:]
return acov

To sum up: For a given time series \((x_0, x_1, \ldots, x_n)\) and a lag \(h\) and of the list, the resulting formula for unbiased autocorrelations is given by:

\[
r_{\text{unbiased}}(h) = \frac{1}{(n-h)\sigma^2} \cdot \sum_{i=0}^{n-1-h} (x_{i+h} - \mu) \cdot (x_i - \mu)
\]  

\(3.1\)

The biased formula, which is the standard \texttt{acf} formula divides the sum by \(n\sigma^2\) instead of \((n-h)\sigma^2\):

\[
r_{\text{biased}}(h) = \frac{1}{n\sigma^2} \cdot \sum_{i=0}^{n-1-h} (x_{i+h} - \mu) \cdot (x_i - \mu)
\]  

\(3.2\)

The standard deviation \(\sigma^2\) is defined as:

\[
\sigma^2 = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \mu)^2
\]  

\(3.3\)

The difference between the biased and unbiased version of the \texttt{acf} formula is shown in Figure 3.17 and 3.18. With increasing lag \(h\) the autocorrelations and anti-correlations calculated with the biased \texttt{acf} formula decrease, since the amount of summands in the dividend decreases while the divisor remains \(n\). In contrast the autocorrelations and anti-correlations calculated with the unbiased \texttt{acf} formula do not decrease and therefore it is easier to see if a reoccurring pattern appears in a long signal. This is the reason, why this formula will be used for all plots, that will follow this section.
3.8 Significance Test

The visual inspection of the sample autocorrelation has two major drawbacks. First of all, it is inconvenient to inspect autocorrelations from many different samples by hand. Furthermore a manual inspection does not provide any information about the significance of the findings. The Ljung-Box-Test addresses both problems. Since the Ljung-Box test uses the $\chi^2$ distribution, it is introduced in the next section. Afterwards the Ljung-Box test is described.

3.8.1 The $\chi^2$ Distribution

Let $Z_1, Z_2, ..., Z_k$ be $k$ different independent standard normal random variables where $Z_i \sim \mathcal{N}(0, 1)$ and let $X$ be sum of the squares of these random variables $X = \sum_{i=1}^{n} Z_i^2$. Then $X$ has a $\chi^2$ distribution with $n$ degrees of freedom, which is denoted by $X \sim \chi^2_n$ [32].

The pdf of $\chi^2_k$ of is given by [32]:

$$f_X(x) = \begin{cases} \frac{1}{2^n n/2 \Gamma(n/2)} x^{n/2 - 1} e^{-x/2} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

The symbol $\Gamma$ refers to the gamma-function. Figure 3.19 gives the $\chi^2$ distributions for different degrees of freedom [32].

![Figure 3.19: The $\chi^2$ distribution for $n \in \{1, 2, 4, 6\}$ [32]](image)

3.8.2 The Ljung-Box Statistics

The null hypothesis of the Ljung-Box test states, that the given timing series does not contain any correlations, so the autocorrelation of all lags are zero or close to zero. The Ljung-Box test takes the autocorrelations of the first $k$ lags from a sample into account in order to check, if the Ljung-Box statistics $Q(k)$ is asymptotically $\chi^2$ distributed with $k$ degrees of freedom [33].
The test is performed as follows: First of all the Ljung-Box statistics \( Q(k) \) is calculated for the autocorrelations of the first \( k \) lags, which is given by the formula

\[
Q(k) = n(n + 2) \sum_{h=1}^{k} \frac{r(h)^2}{n - h}
\]  

(3.5)

where the amount of lags \( k \) is determined by the authors a-priori knowledge. Afterwards, the critical value of the \( \chi^2 \) distribution for \( k \) degrees of freedom and a significance level \( \alpha \) is looked up in a table that can be found in many formularies and compared to \( Q(k) \). If \( Q(k) \) is smaller than the critical value, then the null hypothesis is satisfied [33].

The method \texttt{acf} from \texttt{statsmodels.tsa.stattools} optionally returns the results of the Ljung-Box test, if the parameter \texttt{qstat=True} is given in addition. It then returns the tuple \texttt{acf, qstat, pvalues}, where \texttt{acf} is an array containing the autocorrelations of the first \( k \) lags, \texttt{qstat} is an array that contains the Q-statistics \( Q(k) \) for these lags and \texttt{pvalues} is an array which gives the p-values of the Ljung-Box statistics. Therefore it is not necessary to look up the critical values for \( Q(k) \), but instead the p-values can be used. If the \texttt{pvalue} is smaller than the significance level \( \alpha \), which is usually set to 0.05, then the null-hypothesis is rejected. The Ljung-Box test has been extremely sensitive towards any periodic noise during the experiments, therefore the unbiased version of the autocorrelation formula is applied in order to receive large sums quickly so that the null hypothesis is rejected less often, which means that the number of potential correlations found within different signals is reduced.

3.8.3 Adjusting \( \alpha \)

If just a single test is performed, then \( \alpha = 0.05 \) ensures a probability of 0.05 of getting a false positive result. However if multiple tests are performed, this value needs to be adjusted, because chances of getting at least one false positive result increase. If \( m \) tests are performed, the chances of getting at least one false positive results is \( 1 - (1 - \alpha)^m \). Therefore for two tests, the probability of getting at least one false positive result is 0.0975, for ten tests it is about 0.64 and for 100 tests it is about 0.99 [34].

One way to deal with this problem is given by Bonferroni’s correction, where the target significance threshold \( \overline{\alpha} \) is divided by the number of tests being performed [34]:

\[
\overline{\alpha} = \frac{\alpha}{m}
\]  

(3.6)

This method has some drawbacks. Controlling the Type I errors this way increases the likelihood of Type II errors, which is likelihood of the occurrence of false negatives [35]. However since the Ljung-Box test was quite sensitive during my experiments, I preferred this method from more powerful ones like the fdr.

3.8.4 Constructing Input Signals for the Autocorrelation Method

In order to use the autocorrelation method, an input signal is required. These input signals can be constructed from the measurement results.

Let \( T_x(k, m) \) be the measured execution time of a message \( m \), which is encrypted with
key $k$. Since the encryption times are measured several times for the same key and the same message, $x$ indicates the iteration, in which the time $t_x$ was measured. One way to construct a signal $S_{k_0}$ with potentially repeating pattern is to take the specific key $k_0$ and several messages, for instance $m_0, m_1, m_2, m_3$ in order to perform several measurements. Each of these messages is encrypted with key $k_0$ three times. These results are then brought into the following order:

$$S_{k_0} = \{T_0(k_0, m_0), T_0(k_0, m_1), T_0(k_0, m_2), T_0(k_0, m_3),$$
$$T_1(k_0, m_0), T_1(k_0, m_1), T_1(k_0, m_2), T_1(k_0, m_3),$$
$$T_2(k_0, m_0), T_2(k_0, m_1), T_2(k_0, m_2), T_2(k_0, m_3)\}$$

Under perfect conditions, where no noise occurs, the following formulas hold, since the time required for encrypting a specific message with a specific key is always the same:

$$a = T_0(k_0, m_0) = T_1(k_0, m_0) = T_2(k_0, m_0)$$

$$b = T_0(k_0, m_1) = T_1(k_0, m_1) = T_2(k_0, m_1)$$

$$c = T_0(k_0, m_2) = T_1(k_0, m_2) = T_2(k_0, m_2)$$

$$d = T_0(k_0, m_3) = T_1(k_0, m_3) = T_2(k_0, m_3)$$

Therefore the signal $S_{k_0}$ could also be represented by:

$$S_{k_0} = (a, b, c, d, a, b, c, d, a, b, c, d)$$

So there is a pattern $(a, b, c, d)$, which repeats three times. If $a \neq b \neq c \neq d$, then this pattern will be detected by the autocorrelation method. The expected results are high autocorrelations at lag 4 and at lag 8, where the pattern repeats. Otherwise if $a = b = c = d$ there will be no autocorrelation detected.

Basically there are two questions, that need to be answered by this and the next experiments. The first question is, whether there are any pattern, that can be detected and the second question is, how much the results are influenced by noise.

In order to analyze the results, I also use a second method to construct a signal $Sm_0$. Here the message $m_0$ is kept constant and several keys $k_0, k_1, k_2, k_3$ are used to encrypt that message:

$$S_{m_0} = \{T_0(k_0, m_0), T_0(k_1, m_0), T_0(k_2, m_0), T_0(k_3, m_0),$$
$$T_1(k_0, m_0), T_1(k_1, m_0), T_1(k_2, m_0), T_1(k_3, m_0),$$
$$T_2(k_0, m_0), T_2(k_1, m_0), T_2(k_2, m_0), T_2(k_3, m_0)\}$$

This kind of signal will be used as too, in order to analyze the measurement results. In contrast to the last signal $S_{k_0}$, which can only be used to find leaks caused by different message blocks, the signal $S_{m_0}$ can be used to find leaks caused by keys.

### 3.9 Measuring a Baseline

Before measuring and autocorrelating the execution times of the encryptions, a more simple experiment is performed.
3.9.1 Setup

This experiment uses the loop structure, which will also be used within the next experiments, where the execution times of encryptions are measured. Firstly my algorithm iterates over the keys, within that loop it iterates over the number of iterations and within that loop it iterates over the number of message blocks. The number of keys is set to 20, the number of blocks is set to 50 and the number of iterations is set to 401. This sizing also corresponds to the next experiments. The main difference to the other experiments is, that no keys or message blocks are used. After the two inner loops have finished, all measurement results are printed to the ringbuffer.

There are two volatile variables in this experiment, one is called result and one is called spaceholder. The spaceholder is assigned to a specific value at the beginning of the key number loop, where usually the next key is chosen and initialized. Furthermore spaceholder is assigned to another value at the beginning of the iteration number loop, where usually the first message block is selected. The spaceholder variable is entirely used to preserve the loop structure. The result value is assigned to a specific integer value within the message block loop. Afterwards a number of additions with other integers is applied to result. The corresponding integers have been assigned to fixed values before any of the loops is called. These values do not change during the experiment. However the number of integers which is used for the additions varies with the number of additions. I made separate experiments with 4, 5, 6, and 7 additions. In contrast to the next experiments none of the input values is kept inside an array.

3.9.2 Results

Figure 3.20 presents the resulting autocorrelation charts, when four additions are performed. Although the figure uses the term key and the term block, no real keys or blocks have been used within this experiment. However the measurement results are ordered as if they would correspond to keys and message blocks in order to construct artificial signals.

Many keys show a pattern similar to key 17 in figure 3.20. However the re-occurrences of the higher peaks do not exactly match the loop structure. In contrast the blocks show a clear pattern within the autocorrelation chart.

![Figure 3.20: Autocorrelations for four additions - 400 iterations in total - only the first 20 iterations are presented](image-url)
Adding just a single addition inside that loop results in very different pattern. This is presented in figure 3.21.

Figure 3.21: Autocorrelations for five additions - 400 iterations in total - only the first 20 iterations are presented

Adding another addition leads to the results presented by figure 3.22.

Figure 3.22: Autocorrelations for six additions - 400 iterations in total - only the first 20 iterations are presented

Adding another addition changes the detected pattern again. Figure 3.23 shows the autocorrelations for seven additions.

Figure 3.23: Autocorrelations for seven additions - 400 iterations in total - only the first 20 iterations are presented
3.9.3 Discussion and Conclusion

A number of repeatedly executed instructions often cause some pattern, which can be detected with the autocorrelation method. The pattern itself depends on the instructions. Under these circumstances it is quite difficult to find a proper baseline.

3.10 Autocorrelating the Measurement Results in C

The goal of the next experiment is to use the autocorrelation method to find a pattern that indicates potential leaks from the execution times. In order to ensure that potential leaks do not arise from the measurement method itself, two different measurement strategies are used.

3.10.1 Setup

In this experiment, 20 keys will be used to encrypt 50 message blocks 400 times. A larger setup was not possible, because the ringbuffer and the amount of heap memory required by the arrays need a lot of resources. Furthermore, the autocorrelation function consumes a lot of resources too and therefore the number of measurement results which can serve as input values for the autocorrelation function is also limited.

The first key is the "Sixteen byte key", the second key only consists of zeros and the third key only consists of ones. The other seventeen keys consist of randomly generated bytes. The first message block is "Attack at dawn!!", the second message block only consists of zeros and the third message block only consists of ones, while the other 47 blocks again consist of randomly generated bytes.

Two approaches are used in this experiment in order to measure the encryption times. The first approach takes the whole list of message blocks and encrypts them one after the other. This is done 401 times for each key. The second approach takes a single message block and encrypts it 401 times before the next message block is encrypted 401 times. This procedure is applied to every single key. The first setup might contain some noise from the cache, because each block is accessed once and afterwards, another block is accessed, which has to be loaded first. The second approach simply takes the same block several times, so that new message blocks do not have to be loaded all the time.

The encryption times are saved to a large array, while the message blocks are encrypted by a specific key several times. After all iterations over all message blocks have finished for a specific key, the content of the array cells are printed to the ringbuffer. This is required, because the amount of heap is limited. Printing the results directly after each measurement in contrast can cause a lot of noise within the measurements. This is the reason, why the current strategy has been chosen. Using a cpuid instruction after the loop, which prints out the results, also reduces the noise. Further experiments would be necessary to determine, if this happens, because of the barrier characteristics of cpuid or because of its overhead, that causes a time delay.

The results, which are read from the log file after all measurements have finished, are then sorted in a way, that there is always a pair of a key and a list. The list contains 401
sublists, since there are 401 iterations. Each of these sublists contains the encryption times of all 50 blocks. The results from the first iteration are dropped before the analysis, because the encryption times of the first iteration are always a lot higher compared to the encryption times of other iterations.

To check, if a signal can be detected, when the same key decrypts the same message, I construct a signal similar to \( S_{k_0} \) in the previous section. The 20 last iterations of a key encrypting 50 message blocks are taken and the measured timings are collected in a list. This signal is then used for autocorrelation. A peak should occur every 50 lags in the autocorrelation plot, if timing information is leaked, because this is, when the same message is repeated.

The next step is to check, if any block leaks any information about the key, so I construct a signal similar to \( S_{m_0} \) from the previous section. Therefore the measurement results need to be reordered. A specific message block is chosen, and then the encryption time of the first iteration from the considered 20 iterations from the first key is taken and added to the signal trace. Afterwards the encryption time of the second key is selected and added to the signal trace. This continues for all keys. If the first encryption time of that block has been added for all keys, the same procedure is repeated for the next iteration. Afterwards autocorrelation is applied to the resulting list.

The reason for only presenting the last 20 iterations is that the autocorrelation plots become too large for visual evaluation otherwise.

### 3.10.2 Results

The first plots present the results from the first setup, where a different block is encrypted each iteration. Figure 3.24 shows the autocorrelation charts of the last 20 iterations over all 50 message blocks for key 0, key 11, key 18 and key 19. For key 18 and 19, high autocorrelations are detected every 100 lags and even higher autocorrelations are detected every 200 lags. Smaller autocorrelations are also detected at a frequency, which does not match the occurrence of the blocks at all. The charts of key 14 to key 17 are looking quite similar. The autocorrelation plots of the other keys show way less peaks and look similar to key 0 and key 11.
Even though key 18 shows high autocorrelations, the detected patterns are hardly visible in the input signal. Figure 3.25 shows the last 20 iterations of key 18, which is used to encrypt the blocks.

If the sample size is increased and the first 1000 lags of all 400 iterations are plotted, most autocorrelations become smaller and the pattern becomes more even. The effect is shown in Figure 3.26. If the measurement results of a specific block and a specific key are shuffled, the autocorrelations are becoming even smaller. Shuffling is a good way to check for signals, since it makes the measurement results more independent from each other and from temporal effects like system noise, that may affect a row of measurement results but stop at some point. Most plots show no clear peaks every fifty blocks, some of them show very small ones, eg. key 18 in Figure 3.27.
The next plots represent the results of the artificial signal, where the message block is kept constant and different keys are used. Some of these blocks show a pattern every 80 lags, which are probably caused by the same effect, which was already observed in the key traces. Key 38 shows a pattern every 40 lags and Key 17 and key 20 show a weak 20-lag pattern. Clear leaks every 20 lags have however not been found with this test. Some selected plots from the message blocks are presented in Figure 3.28.

Increasing the sample size leads again to a pattern every 80 lags for most keys, and just a few charts show a pattern every 20 lags. Examples are given by figure 3.29.
In order to reduce potential underlying temporal effects, I also shuffled the measurement results of each key-message pair. Therefore I took all measured execution times from a specific key and from a specific message and shuffled their order. I did this for each key and message pair separately. Afterwards I constructed the artificial signals again. The resulting autocorrelation plots have no or only extremely small peaks every 20 blocks and look similar to the plots in Figure 3.30.

The Ljung-Box-Test is carried out the following way: The Ljung-Box-Test is applied to all lags from one to 100 for traces, where the key is kept constant and it is applied to all lags from one to 40 for traces, where the block is kept constant. My significance threshold is 0.05, since I am testing 20 keys and 50 message blocks. Block 2 presented in 3.30 does not show any leaks in contrast to Block 27 from the same figure. Table 3.7 shows the results for all 50 block autocorrelation traces. Many leaks already occur before lag 20, which is before the first repetition occurs. I would expect the first leak to appear at lag 20, because this is where the first large autocorrelation would occur, if a timing leak exists. However existent signals usually also have some correlation or anticorrelations with themselves at any lag, which can be observed in the plots given in the autocorrelation section. So either the Ljung-Box-Test detected periodic noise or it shows these small correlations and anti-correlations within the signal itself. Leaks, which start at lag 20 are quite rare, no matter if just the last 20 iterations or all 400 iterations or the shuffled 400 iterations were taken into account. Leaks were however more often detected, when all 400 iterations were used, and when they were not shuffled. Furthermore they already occur for very small lags. The results for the
autocorrelation traces, where the key is kept constant look quite similar. Table 3.8 shows, that most Ljung-Box-Test start to show potential leaks before lag 50 is reached. This is where the blocks are repeated and the first large autocorrelation would occur within the sum, if a timing leak exists.

<table>
<thead>
<tr>
<th></th>
<th>first leak &lt; lag 50</th>
<th>first leak = lag 50</th>
<th>first leak &gt; lag 50</th>
<th>no leak</th>
</tr>
</thead>
<tbody>
<tr>
<td>last 20 iterations</td>
<td>8</td>
<td>2</td>
<td>9</td>
<td>31</td>
</tr>
<tr>
<td>400 iterations</td>
<td>36</td>
<td>7</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>(shuffled)</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>35</td>
</tr>
</tbody>
</table>

Table 3.7: Ljung-Box Test results for autocorrelations with constant blocks

In comparison to the autocorrelations detected in the first setup of the experiment, the results from the second setup show much weaker autocorrelations. The autocorrelations of the same keys that were presented in the previous plot are given by figure 3.31. The trace of key 0 and key 18 are the only ones, that show autocorrelations every 50 lags, which are higher then the autocorrelated background noises. Key 13 shows some weak autocorrelations every 50 lags and whereas the autocorrelation plot for key 14 does not show any visible pattern. Most plots look similar to the one from key 14 or like a mixture of the autocorrelation plots of key 13 and 14.
Figure 3.31: Autocorrelations of Different Keys - 20 Iterations

The trace of key 0, which is given by figure 3.32 shows, that the execution time differences are again very small and probably not big enough for an attacker to exploit these, especially if some noise from the system is added.

Figure 3.32: Signal of Key 0

Increasing the sample size and shuffling the iterations still leads to large autocorrelations for key 0 and key 18 in Figure 3.33. All other keys have smaller autocorrelations, similar to key 13 and key 14.
In contrast, only some of the autocorrelation plots, where the message blocks are kept constant, show some small autocorrelation pattern, for instance block 13 in figure 3.34, but they are the minority. Most plots just show a lot of noise.
Figure 3.35 shows, that increasing the sample size and shuffling the iterations results in small autocorrelation patterns, which appear every 20th lag. The highest autocorrelations appear in the plot of block 13, whereas some blocks like block 49 only show noise.

Figure 3.35: Autocorrelations of Different Blocks - 400 iterations in total (shuffled) - only the first 20 iterations are presented

For this setup of the experiment, the Ljung-Box-Test is applied again. The first 100 lags from the traces where the keys are constant are tested and the first 40 lags from the traces, where the blocks are constant are tested too. The threshold is again $0.05$. Table 3.9 and table 3.10 show the results. Again the test turns positive already at the very first lags. Even the autocorrelation plot of key 14 in Figure 3.33 has been tested positive for potential leaks, although there is hardly any signal visible in the plot.

One aspect needs to be considered, when the Ljung-Box-Test results are compared to

<table>
<thead>
<tr>
<th></th>
<th>first leak &lt; lag 20</th>
<th>first leak = lag 20</th>
<th>first leak &gt; lag 20</th>
<th>no leak</th>
</tr>
</thead>
<tbody>
<tr>
<td>last 20 iterations</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>400 iterations</td>
<td>49</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>400 iterations (shuffled)</td>
<td>33</td>
<td>9</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3.9: Ljung-Box-Test results for autocorrelations with constant blocks

the charts: The exact order of the shuffled results from the plots and from the Ljung-Box-Test are slightly different. The reason for this is, that the shuffling method is called for the plots and for the Ljung-Box-Test independently from each other. However the autocorrelation pattern of the shuffled charts are very even, so therefore the results of the Ljung-Box-Tests should still be meaningful.
3.11. AES MEASUREMENTS IN USERSPACE

<table>
<thead>
<tr>
<th></th>
<th>first leak &lt; lag 50</th>
<th>first leak = lag 50</th>
<th>first leak &gt; lag 50</th>
<th>no leak</th>
</tr>
</thead>
<tbody>
<tr>
<td>last 20 iterations</td>
<td>12</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>400 iterations</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(shuffled)</td>
<td>17</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.10: Ljung-Box-Test results for autocorrelations with constant keys

3.10.3 Discussion and Conclusion

The autocorrelation method detects small timing differences, when different keys encrypt different blocks. However further research is required in order to ensure, that the detected signals arise from the encryption algorithm and not from the measurement method itself or some system noise. The autocorrelation pattern detected in the first setup of the experiment shows, that high autocorrelations can be introduced by the measurement method itself. However small autocorrelations, that might show potential leaks have been detected in both setups. Fortunately the variances of the encryption execution times are very small, therefore it is very likely, that the detected differences cannot be measured in user space, where additional noise is introduced, since preemption and hard interrupts cannot be switched off then.

Autocorrelations can be visualized better, when the samples that consist of the measured execution times from a specific key and a specific block are shuffled, before they are plotted. Larger samples also improve the quality of an autocorrelation plot.

The Ljung-Box test did not perform well in this experiment. It seems to be oversensitive towards small autocorrelations and it cannot distinguish, if the there is a pattern, that would fit to the number of keys or to the number of blocks.

3.11 AES Measurements in Userspace

Before applying the autocorrelation method to the OpenSSL library, which is vulnerable to timing attacks, a short experiment in users mode is performed in order to get an idea about the amount of noise which has to be expected. Furthermore measuring of execution times in kernel mode is probably not the most practical method. First of all the kernel code is compiled with different options and different libraries. Another potential problem might be, that the person, who runs these tests may not have enough privileges on the target platform in order to run the application in kernel mode.

3.11.1 Setup

In this experiment, the previously used PyCrypto AES encryption is used again. Because it nearly performs in constant time, it is a good tool to visualize noise. Since the second setup from the previous experiment seemed to be less affected by system noise, it is used for this experiment too.

In order to execute the test program in user mode, all printk statements are replaced by printf, all kmalloc statements are replaced by malloc and all kfree statements are replaced by free. Another difference to the previous experiment is that according to the
disassembled code, the encryption function has not been inlined. Since hardly any library offers an encryption function, which can be inlined, a function call might be more practical in this case.

Again 20 keys are used to encrypt 50 blocks 401 times. Each block is encrypted 401 times, before the next block is encrypted 401 times. During the experiment, the programs runs on its own virtual and physical core. The second virtual core from that physical core is not supposed to be used. The operating systems runs on the other physical core.

The keys and the data that are used for the encryptions are the same as in the previous experiment. The evaluation is also done exactly the same way as previously.

### 3.11.2 Results

Within the previous experiment, using all 400 iterations and shuffling looked most promising, therefore this method is applied to these results too. Figure 3.36 presents trace of 400 encryptions of key 0 and block 8 in user space and in kernel space. There is a lot more noise in user space and some of the noise appears to be periodic or nearly periodic. Furthermore the measured execution times are about two times higher in user mode compared to the kernel mode.

![Figure 3.36: Execution times of Key 0 Block 8 in User Mode (blue) and in Kernel Mode (red)](image)

According to the autocorrelation method, the encryption algorithm runs nearly in constant time. Extremely small autocorrelation patterns can be detected within the autocorrelation plot of key 4 in Figure 3.37. However this is the only key, where the pattern appears. Other keys might produce a single autocorrelation peak at a single lag, which is a multiple of the amount of blocks that are encrypted, but these peaks could also be caused by noise.
Regarding the autocorrelation plots, where the blocks are kept constant, most autocorrelation plots look like those of block 1 and block 3. Block 16, 17 and 38 are the only blocks which contain a few higher autocorrelations at the multiples of the number of keys.
The Ljung-Box test in this experiment shows way less potential leaks than in the previous experiment. Only three out of 20 autocorrelation signals show potential leaks, when the key is kept constant and the iterations of the key message pairs are shuffled. Furthermore only six out of 50 autocorrelation signals show potential leaks, when the message block is kept constant and the iterations of the key message pairs are shuffled.

### 3.11.3 Discussion and Conclusion

The additional noise from user space efficiently hides the signal coming from the encryptions, at least if the attacker tries to discover it using the autocorrelation method or the Ljung-Box test. Although more measurements are affected by noise from the system, it is still possible to see the small zick-zack pattern that is caused by the nearly in constant time performing encryptions and which has also been visible within the previous experiment.

### 3.12 Experiments with the exploitable OpenSSL RSA decryption

In this experiment the autocorrelation method is applied to the RSA decryption implementation provided by OpenSSL\(^3\), which is exploitable with a timing attack if blinding is disabled \[9\]. OpenSSL is a popular open source cryptography and SSL/TSL Toolkit. Some of its modules have been FIPS-140-2 certified, which confirms that they comply with the security requirements for cryptographic modules from the US American National Institute of

\(^3\)https://www.openssl.org/
Standards and Technology (NIST) [36]. The goal of the next experiments is to use the autocorrelation method and the Ljung-Box-Test in order to find the existing timing leaks.

### 3.12.1 Setup - Experiment I

In contrast to the previous experiment, the library is pre-compiled and therefore it just needs to be linked to the test application. This might introduce additional noise but makes it again more practical, since this is the way how most libraries are used. Furthermore this time the decryption time is measured and therefore each block has to be encrypted, before it can be decrypted. Another change to the previous setups is, that instead of a large array, a file is used, which contains the input data. AES-128 is only able to encrypt 16 bytes of data with a 128 bit key. In contrast RSA-2048 can decrypt 256 bytes of data, if no padding is used, since the keys, which are used within this experiment are 2048 bits long. As a result a lot more input data is required.

All RSA .pem keyfiles for public and private keys have generated using OpenSSL from the command line. The input data has been generated using Python. The data for the second block only contains zeros and the data for the third block only contains ones. All other blocks consist of random bytes, which were generated with `os.random`.

The setup of this experiment is quite similar to the setup from the last experiment. 20 keys encrypt 50 blocks 401 times. The measurements are done block by block, therefore each block is encrypted once and then it is decrypted 401 times, before the next block is encrypted and then decrypted 401 times. This procedure is applied to two different scenarios. In the first setup the blinding option is disabled, whereas in the second setup the blinding option is enabled.

### 3.12.2 Results - Experiment I

When blinding is disabled, some of the resulting autocorrelation plots where the keys are kept constant show quite large autocorrelations eg. key 2 in Figure 3.39, whereas some autocorrelation plots show only very small autocorrelations, for instance key 11. In total 15 out of 20 autocorrelation plots, where a key is kept constant show autocorrelation patterns with autocorrelations above 0.25.
The autocorrelation patterns of the autocorrelation plots, where the blocks are kept constant, are even larger. All plots except from those of block 9 and block 44 show autocorrelations above 0.75. These plots are shown in Figure 3.40.
Figure 3.40: Autocorrelations of Different Blocks - no Blinding (400 iterations in total - shuffled - only the first 30 autocorrelations are shown)

Figure 3.41 shows the traces of key 0 and key 2 in exactly the order, in which the execution times were measured, so that each block is decrypted 400 times, after the previous block has been decrypted 400 times. There are some measurements, which take constantly more time than others. The thin vertical lines indicate where a new block begins. It can be seen that the jump to a higher or lower level often occurs, while the same block is decrypted, therefore this effect must be caused by some underlying noise from the system. It also shows, that the measurement results are not independent from each other.
Figure 3.41: Timing traces of key 0 and key 2

When the measurement results are sorted according to a key or a block, the autocorrelation method detects the underlying noise instead of a true signal. Shuffling seems to be a potential way to diminish this effect and therefore the autocorrelations in the plots, where the results have been shuffled are higher than the autocorrelations in the plots, where the results have not been shuffled. This effect can be observed at the first autocorrelations of key 11 and block 9 in Figure 3.42, where key 11 shows way higher autocorrelations than in figure 3.39 and block 9 shows way higher autocorrelations than in figure 3.40.

Figure 3.42: Autocorrelations of Different Blocks - no Blinding (400 iterations in total - not shuffled - only the first 30 iterations are shown)

The Ljung-Box tests show leaks for all autocorrelation traces. Most leaks already appear at lag 1, no matter if the key or the block is kept constant and no matter if the measurement results have been shuffled or not.

In contrast to the previous setup the autocorrelation plots of the blinded version given by figure 3.43 show no pattern at all for any key or any block. The Ljung-Box test also detects no keys and no blocks, which potentially leak information.
Figure 3.43: Autocorrelations of unblinded Keys and Blocks (400 iterations in total - shuffled - only the first 30 iterations are shown)

These test results do not prove, that the blinding method works efficiently. The un-shuffled versions of the autocorrelation plots in Figure 3.44 show a high autocorrelation pattern every 32 lags for all keys but hardly any autocorrelations for the blocks. The Ljung-Box test also looks different for the un-shuffled version: It shows potential leaks for all keys and all blocks in the un-shuffled version, most of them starting at lag 1.

Figure 3.44: Autocorrelations of unblinded Keys and Blocks (400 iterations in total - shuffled - only the first 30 iterations are shown)

The detected pattern arise from an implementation detail of OpenSSL’s RSA blinding algorithm. Every 32 decryptions, new blinding parameters are created [37]. Figure 3.45 shows one of the timing traces, where a appears peak every 32 encryptions.

Figure 3.45: Timings of Key 0 Block 2
The traces from the keys still show jumps between two different levels, but they are very small compared to the amount of time, which is required by the re-creation of the blinding parameters. As an example, figure 3.46 presents the trace of key 4, where the jumps between the levels can be observed at about 2,000,000 cycles and where the parameter recreation takes at about 3,400,000 cycles.

![Figure 3.46: Trace of Key 4](image)

### 3.12.3 Discussion and Conclusion - Experiment I

The autocorrelation method was able to discover a 32 lag pattern, which was caused by the noise from the creation of new blinding parameters. The shuffling method showed, that it is possible to get rid of these pattern that way. However it remains unclear, if an autocorrelation pattern would appear, when the noise from the recreation process is filtered. The Ljung-Box-Test performed better this time, it only showed a small number of false-positives. Another observation during this test is, that the measurement results are not independent from each other.

### 3.12.4 Setup - Experiment II

The last experiment has not been suitable to evaluate the quality of the blinding approach. Furthermore it was also not possible to distinguish timing differences caused by the library from timing differences caused by underlying system noise. Therefore a second experiment is performed, which follows a different approach, which is similar to the first setup from the kernel space AES test. Each block is encrypted by a key and then it is decrypted 8 times. The time of the last execution is taken and then the next block is decrypted 8 times. This is done in order to warm-up the system. Furthermore this approach also avoids, that the re-creation of blinding parameters shows up in the results and it ensures, that different blinding parameters are used for the same block. The whole measurement procedure is repeated 201 times for each block and each key. The experiment is performed with the blinded and the unblinded RSA implementation.

### 3.12.5 Results - Experiment II

The autocorrelation plots where the blocks are kept constant look all similar and the autocorrelation plots where the key is kept constant look all similar, when the unblinded RSA version is used. Therefore Figure 3.47 presents just a single key and a single block. While the autocorrelation charts of the keys look like random noise, the autocorrelation charts of the blocks show a clear pattern which fits to the keys, that are used to decrypt that specific block.
Figure 3.47: Autocorrelations of unblinded keys and blocks (200 iterations in total - shuffled - only the first 30 iterations are shown)

The un-shuffled versions in Figure 3.48 show hardly any autocorrelation pattern for the key traces but a nearly similar pattern for the blocks.

Figure 3.48: Autocorrelations of unblinded keys and blocks (200 iterations in total - not shuffled - only the first 30 iterations are shown)

The autocorrelation plots from the decryptions which used blinding don’t show any autocorrelation pattern, when the key is kept constant, while they do, when the blocks are kept constant.

Again all autocorrelation plots where the key is kept constant look very similar. An example is given by the plot of key 0 in figure 3.49. The only exception is key 8.

Figure 3.49: Autocorrelations of blinded keys (200 iterations in total - shuffled - only the first 30 iterations are presented)
Extremely weak autocorrelation patterns can be found in some of the key traces of the unshuffled keys in figure 3.50 too, but they do not appear every 50 lags. Therefore it is probably just noise.

The autocorrelation plots where the blocks are kept constant look all similar to each other. The autocorrelation plot of block 1 in Figure 3.51 is given as an example. In contrast, all un-shuffled versions look like block 1 in Figure 3.52.

The traces of the blinded and of the unblinded version show again, that there seem to be two levels of speed, at which the decryptions are executed and which do not seem to depend on the keys or blocks. Figure 3.53 shows this effect for key 7.
According to the Ljung-Box test, one out of 20 autocorrelation signals where the key is kept constant may contain a potential leak when blinding is disabled and two out of 20 autocorrelation signals may contain a potential leak, when blinding is enabled. Furthermore all 50 autocorrelation signals where the block is kept constant show a potential leak, no matter if blinding has been used or not. Again most leaks start at lag one. If the measurement results are not shuffled, then all keys and all message blocks show potential leaks.

3.12.6 Discussion and Conclusion

This measurement method could also not provide a clear result about potential leaks. It is very likely, that the detected autocorrelation patterns arise from the two different levels of execution speed, that have been observed. The reordering of the measurement results might be the reason, why no pattern is detected in the traces, where the key is constant but in all traces, where the block is kept constant. Ongoing noise can be filtered, but if the measurement results are mixed the way I did it in order to keep the blocks constant, then the noise is suddenly re-occurring and therefore it can be detected by the autocorrelation function which leads to patterns, that indicate leaks. For instance if a key A has mainly a rather slow execution time, whereas another key B has a mainly fast execution time and these keys are reordered, so that the different measurements from key A and key B alternate, a pattern will be detected by the autocorrelation method, although there are just the different levels of noise, which are detected. In contrast, if the pattern like 3.53 appears, where a series of slow execution times is followed by a series of large execution times, the signal cannot be found again, at least not, if only small differences within the execution times are present. Therefore the autocorrelation method was able to detect the 32 lag pattern from the blinding recreation, however the difference of the execution times was quite large compared to the different execution time levels. Potential execution time variances are probably quite small, if they exist. Increasing the sample size will probably not work, since only more jumps between the execution time levels would be introduced.

The main difference between the blinded and the unblinded RSA decryption, that has been detected in this experiment is, that the autocorrelation pattern of the blocks are slightly above 0.5, when blinding is switched off, whereas the autocorrelation pattern of the blocks with blinding are slightly below 0.5. The autocorrelation pattern for the traces with constant keys look quite similar, no matter if blinding is used or not.

The Ljung-Box test has shown the similar results for both, the blinded and the unblinded version. Therefore it has also not successfully detected a difference.
To sum up, it was not really possible to distinguish the blinded from the unblinded version of OpenSSL's RSA decryption in this experiment. Furthermore it was not possible to detect the existing leak easily.
Chapter 4

Summary

In order to find timing differences in the PyCrypto implementation of AES, various benchmarking methods in Python have been tried. The most reliable one seems to be `time.time()`. Using less cores or the FIFO scheduling strategy did not significantly improve the results.

A more precise method is provided by Intels RDTSC/RDTSCP instructions. However it is difficult to find a good classifier for this method, since minimal execution times, which worked well for Python, are usually outliers when this measurement method is applied. Furthermore some extremely high outliers strongly bias the median and the average. For this type of measurement it was very beneficial, if the test program was executed on a single virtual and physical core. If the execution times were measured in kernel mode, they were a lot lower than in user mode. However the small zigzag pattern, which was visible in the execution time trace from the experiment in kernel mode was still visible in user mode.

Although autocorrelation might be a good method to find signals, it turned out to be very sensitive especially to periodic system noise. Therefore high autocorrelations often do not indicate any key or message dependent pattern but instead they indicate periodic noise. Furthermore this method does not show, if the detected execution time differences are rather small or rather large.

Small execution time variances for PyCrypto’s AES algorithm in C have been discovered in kernel space but they vanish in user space, at least when the autocorrelation method is used.

In addition the autocorrelation method has been applied in order to discover timing variances in OpenSSL’s RSA implementation. The fact, that blinding is only refreshed every 32 encryptions or decryptions has been visible immediately. Unfortunately the decryption of a block with different keys always leads to well visible pattern in the autocorrelation plot, no matter if blinding is applied or not. It is unclear, weather this arises from system noise, the measurement method itself or from a vulnerability.

Changing the setup did also not help to detect differences from the execution times of the blinded and the unblinded version of RSA. The underlying noise seems to be very dominant and as a result noise is detected instead of a signal from the decryption.

The Ljung-Box test has been used in order to have some sort of significance test and in order to reduce the amounts of autocorrelation plots, that have to be evaluated manually. Unfortunately the Ljung-Box test only shows the amount of noise up to a specific lag. It does
not show, if a pattern equal to the repetitions of keys or blocks is detected or if just periodic system noise is detected. Furthermore the Ljung-Box test is quite sensitive and produces many false positives.
Chapter 5

Discussion and Conclusion

The experiments confirm, that a precise measurement method is crucial to detect small timing differences. Noise from the system offers some protection against timing attacks, but it also make it more difficult to determine, if a library contains information leaks or not. Some languages like Python do not offer a high precision measurement method, which makes it also very difficult to detect leaks.

Measurements in kernel space are less affected by noise than measurements in user space. However it is questionable, if it is a good idea to test cryptography algorithms in kernel mode, since the compiler uses different flags to compile the code, and therefore the results may not show existing vulnerabilities. Furthermore the person in charge of testing the application might not have enough access rights to the machine, that is supposed to be tested. However the measurements in userspace were also not too unprecise apart from the detected re-occurring nearly periodic noise.

Regarding the execution strategy it is beneficial for the precision of the measurement results, when the test program is executed on a single virtual and physical core. This is especially important in kernel mode.

The autocorrelation method looks promising at first sight, but it is definitely not an easy method to test applications for potential timing leaks. It takes many tries to calibrate these tests regarding the sample size, the number of input values, the input method, the loop structure, etc. and the software tester can never be sure, if the detected patterns arise from the testing method itself, from system noise or from the implementation of the test program.

The Ljung-Box test has been tried in order to automate the evaluation of the autocorrelations. It turned out that the test is oversensitive to noise and gives a large amount of false positives. Furthermore it does not say anything about the pattern, which is expected when a potential leak occurs. Nevertheless using the sums of autocorrelations of a specific amounts of lags may be a good approach. One could try to calculate the average autocorrelations of the lags, where no repetition of keys or blocks appears and compare them to the average autocorrelations of the lags, where the keys or blocks start repeating. Another way to deal with the oversensitivity is to reduce the significance threshold.
5.1 Future Work

The autocorrelation method could be applied to further examples, for instance to the elliptic Curve25519-donna [13], which contains a timing leak. However if the two-level-execution pattern occurs there too, then the results may need to be filtered. Another tool from signaling theory, that might be able to detect potential leaks is the Fourier Analysis.

Regarding the measurement methods, it might be a good idea to write a Python extension, in order to evaluate, if the measurement precision improves, if $\text{RDTSC}(P)$ and $\text{CPUID}$ or the corresponding AMD functions are provided.

Furthermore many processors provide further measurement instructions besides $\text{RDTSC}(P)$. These methods could also be explored and compared to the $\text{RDTSC}(P)$ method.
Bibliography


