FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Timing side-channel attack on AES
Student:	Adam Zahumenský
Supervisor:	Ing. Jiří Buček, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2019/20

Instructions

Study the topic of timing side-channel attacks of the AES cipher on modern CPUs with caches. Measure the times of a naive AES implementation and an AES implementation using T-boxes and analyze the dependence of the calculation time on data, key, and possibly other factors, and try to break the key. The final work will take the form of a laboratory exercise assignment for teaching in computer security subjects.

References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 14, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor’s thesis

Timing side-channel attack on AES

Adam Zahumenský

Department of Information Security
Supervisor: Ing. Jiří Buček, Ph.D.

May 16, 2019

Acknowledgements

I thank my supervisor Jiří Buček for his time and ideas which helped me delve deeper into my discoveries. Furthermore, I thank all of my friends for their unending support when I needed it the most.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 16, 2019 …………………

Czech Technical University in Prague
Faculty of Information Technology
© 2019 Adam Zahumenský. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Zahumenský, Adam. *Timing side-channel attack on AES*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato práce demonstruje časový postranní útok na šifru AES-128 s využitím současného hardwaru. Jejím výsledkem je software, který takový útok využívá k odhalení zranitelnosti v poskytnuté implementaci AES-128. Software je vyvinutý s účelem využití ve výuce počítačové bezpečnosti jako laboratorní úkol. Práce nakonec demonstruje tuto zranitelnost za určitých podmínek v nejnovější verzi OpenSSL.

Klíčová slova AES, postranní útok, časový útok, informační bezpečnost

Abstract

This work demonstrates a timing side-channel attack on the AES-128 cipher using modern hardware. It provides software which leverages such attack to test a provided AES-128 implementation for vulnerability. The software is provided in a form suitable for use as an InfoSec laboratory assignment. Finally, it demonstrates that the latest OpenSSL release is under certain conditions still vulnerable to the attack.

Keywords AES, side-channel attack, timing attack, information security

Contents

List of Figures

List of Tables

Introduction

AES (Advanced Encryption Standard) is one of today's most widely used symmetric ciphers. It features sufficient performance for real-time data encryption, being featured prominently in disk encryption and secure messaging. In this work, I limit my scope to AES-128, an AES variant using a 128-bit key.

The cipher is still considered safe for use with sensitive data. Several cryptographic attacks on the cipher exist with the latest one only about 4x faster than a brute force attack [1]. Given the 2^{128} operations a brute-force attack would require, an attack with such speedup is still completely unfeasible on today's hardware.

Despite the cipher's resistance to cryptographic attacks, side-channel attacks may be used to break specific implementations. Side-channel attacks target implementation flaws and require additional information about the encryption to break the key. In this thesis, the targeted side channel is encryption time.

To protect encryption algorithms from leaking side channels, manufacturers tend to include cryptographic support in hardware, a move proven to be very efficient. Hardware implementations are faster and more secure as the cryptographic operations are moved away from RAM and caches into dedicated hardware and registers. However, hardware support is not omnipresent and software implementations of AES are still critical for systems without such support.

This work aims to present a complete software package able to test any AES-128 implementation for timing side-channel vulnerability. It is provided in the form of a laboratory assignment for students of hardware security at FIT CTU. Besides testing self-written AES implementations I also test the latest OpenSSL build and document my findings.

Chapter 1

AES-128 cipher

1.1 Basic principles

AES-128 is a symmetric block cipher using the substitution-permutation network design. It accepts a 16-byte key k, a 16-byte input block n and outputs a scrambled 16-byte data block.

A symmetric cipher uses the same key for both encryption and decryption. As such, obtaining the key enables an attacker to both read and write data. On the other hand, it simplifies implementations, leading to cheaper dedicated hardware and lower hardware requirements.

A block cipher encrypts data per blocks of fixed size. In the case of AES-128, the block size is 128 bits. Data larger than the block size are then sliced and padded to 16-byte blocks which are then sequentially encrypted and chained using a specific operation mode. One can safely ignore the various cipher operation modes by limiting themselves to repeated encryptions of a single block.

For simplicity, I shall omit AES decryption since this work only uses encryption to gather data.

1.1.1 Encryption

AES-128 works with an internal 16-byte state s initialized to $n \oplus k$ where $\oplus$ denotes the bitwise XOR. It is represented as a 4x4 2D matrix. The key is first expanded to 10 16-byte round keys rk, one for each round, using the Rijndael key schedule [2][section 5.2]. The state is then scrambled in 10 rounds consisting of four steps in this order: *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. The 10th round omits the *MixColumns* step.

AddRoundKey mixes the current round key rk_i into the state: $s' = s \oplus rk_i$.

SubBytes uses a fixed 256-byte substitution table S called an S-box, commonly implemented as a plain array. The state is then substituted byte for byte using the S-box: $s' = S[s]$.

ShiftRows rotates the last three rows to the right: second row by 1, third row by 2 and fourth row by 3.

MixColumns scrambles each column of the state by the following invertible linear transformation:

$$\begin{bmatrix} s'_{0,j} \\ s'_{1,j} \\ s'_{2,j} \\ s'_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix}$$

All bytes above represent polynomials over GF(2^8) of order x^7. All additions are bitwise XOR and multiplications are done modulo $x^8 + x^4 + x^3 + x + 1$.

1.2 T-box optimization

A well-known optimization of the above procedure is to combine the three mentioned steps into one large tabular substitution using four 1024-bit tables known as T-boxes. The following definitions are sourced from [3] and [4]. Each T-box can be viewed as 256 4-byte values defined as follows:

$$T_0[j] = \begin{bmatrix} 02.S[j] \\ S[j] \\ S[j] \\ 03.S[j] \end{bmatrix} \; T_1[j] = \begin{bmatrix} 03.S[j] \\ 02.S[j] \\ S[j] \\ S[j] \end{bmatrix} \; T_2[j] = \begin{bmatrix} S[j] \\ 03.S[j] \\ 02.S[j] \\ S[j] \end{bmatrix} \; T_3[j] = \begin{bmatrix} S[j] \\ S[j] \\ 03.S[j] \\ 02.S[j] \end{bmatrix}$$

Bytes here are again considered polynomials over GF(2^8) as above. The first nine rounds can then be simplified to the following transformation using the round key rk_i.

$$s' = \begin{bmatrix} T_0[s_{0,0}] \oplus T_1[s_{1,1}] \oplus T_2[s_{2,2}] \oplus T_3[s_{3,3}] \oplus rk_i[0] \\ T_0[s_{1,0}] \oplus T_1[s_{2,1}] \oplus T_2[s_{3,2}] \oplus T_3[s_{0,3}] \oplus rk_i[1] \\ T_0[s_{2,0}] \oplus T_1[s_{3,1}] \oplus T_2[s_{0,2}] \oplus T_3[s_{1,3}] \oplus rk_i[2] \\ T_0[s_{3,0}] \oplus T_1[s_{0,1}] \oplus T_2[s_{1,2}] \oplus T_3[s_{2,3}] \oplus rk_i[3] \end{bmatrix} \tag{1.1}$$

AddRoundKey is already included in this transformation, interpreting rk_i as a four 4-byte values. The last round is usually not optimized and simply consists of *SubBytes*, *ShiftRows* and *AddRoundKey*.

Chapter 2

Timing attacks on AES

2.1 Introducing timing side-channel attacks

Side-channel attacks take a different approach from traditional cryptographic attacks. Instead of attacking the cipher mechanism head-on they target weaknesses of specific implementations. Vulnerable implementations may leak information such as electromagnetic waves, power consumption or timing.

This work focuses on the timing side channel, the time an implementation takes to process data. In the case of AES, the simplest timing target is the time taken to encrypt a single block of data.

The only way to make a software AES implementation resistant to leaking timing data is to make it run in constant time. *"It is extremely difficult to write constant-time high-speed AES software for common general-purpose computers."*, Bernstein stated in [4, Abstract]. The performance penalty imposed on constant-time AES software is unacceptable for generic use in demanding applications like disk encryption.

Hardware-supported encryption has the benefit of not using RAM and caches, cutting off access from prying eyes. Moreover, it offers significant performance improvements as lookup tables are hard-wired into the hardware. By extension, this also has the potential to make timing attacks more difficult as caches are not used at all.

2.1.1 Exploiting caches

Every AES implementation needs one or more lookup tables for the substitution steps. In the case of T-box optimized implementations, there is up to 4 kB of lookup data to store in memory.

With recent L1 caches easily hitting hundreds of kB in size, it is expected that the tables will be cached on all levels of the cache hierarchy. As Kocher wrote in 1996, *"RAM cache hits can produce timing characteristics"*[5, section 11]. If certain lines from the lookup tables get kicked out of cache and are accessed later, the miss penalty can be detected. These penalties may help an attack if they are well correlated with controlled variables such as the input block or a known key.

2.2 Bernstein's attack

Bernstein aimed his timing attack at the T-box optimized software implementation of AES-128 in OpenSSL and demonstrated its usability over the network. He based his attack on the following assumption: *"it is extremely difficult to load an array entry in time that does not depend on the entry's index"*[4].

The first round of T-box optimized AES-128 accepts $n \oplus k$ as the input state. Therefore the time of the first T-box lookup should be dependent on bytes of both k and n. As all 16 bytes of s are used as indices in def. 1.1, all bytes of the key could be correlated with timing using the first round alone.

2.2.1 Design

The attack targets a network-accessible victim machine and is split into client and server roles.

The client sends packets of set size to the server which performs encryption using its own secret key k. Bernstein demonstrated his results with 400, 600, and 800-byte packets. Each packet contains random data, and the server only encrypts the first 16 bytes as n. It then echoes back the encrypted block followed by encryption time and the unchanged remainder of the packet.

As a result, each random packet sent equals one encryption by the server plus some extra "work", in this case, the echoing of data back to the client. Bernstein modeled the attack this way to simulate a real server which accepts some work request alongside with additional data. As the server sends back the encryption time, there is no need to gather extra measurements to amount for variable network delays.

The client then tallies the responses for each byte of the cleartext n. The output data of the client has the form of timing statistics for each n byte $[0, 15]$ and its value $[0, 255]$.

The client outputs are later fed to a correlating program that tries to identify possible candidates of k. In the end, a brute-force program tests these candidates in a loop and tries to break k.

2.2.2 Principle

In the first stage, the attacker uses their own machine with an identical CPU to the victim's machine. The key k_1 is known in this case and defaults to zero in Bernstein's implementation, hence $n = k_1 \oplus n$. The attacker then runs the server on this machine and collects timing data for random blocks n_1 using k_1.

The second stage involves the server running on the victim's machine and its secret key k_2. Data collection for random blocks n_2 is the same as above.

With both datasets collected, as both machines use the same CPU and implementation, it is safe to assume that if a correlation exists between $n_1 \oplus k_1$ and encryption time, that correlation[1] is identical for $n_2 \oplus k_2$.

The attacker then goes over all 16 bytes of k_2 and searches for such a value of c which would cause the two datasets, $k_1[i] \oplus n_1[i]$ and $c \oplus n_2[i]$, to correlate the best. These values of c then become the byte candidates for $k_2[i]$.

2.2.3 Results

Bernstein successfully cracked a whole AES-128 key using two machines running Intel Pentium III, now a very old CPU with only two levels of caches. A single run of the attack didn't break every byte but multiple runs with different packet sizes cracked different bytes, eventually revealing enough of the key to run a brute-force attack.

2.3 Further work

Wei Liu et al. tested Bernstein's attack on an Intel Atom N2800, Intel Xeon E5410 and Intel Core i5 with the Atom showing very limited vulnerability, reducing the key space from 2^{128} to 2^{112} [6]. The attack was completely unsuccessful on the latter CPUs but considering the minor keyspace reduction, the Atom was also found practically resistant.

C Ashokkumar et al. used a different type of attack on an Intel Core i3-2100 and Intel Core i7-3770 [7]. They successfully used the cache-based Prime+Probe and Flush+Reload techniques to break an AES-128 key. In this case, the target implementation was not optimized using the T-boxes and instead used the traditional 256-byte S-box. Their work proves that the software implementations of AES are vulnerable to cache attacks even on recent processors. Similar results were achieved by Liwei Zhang et al. who targeted an Intel Core i5 CPU and again used cache-based attacks to extract the key [8].

I found no further successful endeavors using Bernstein's original attack principle.

[1]Bernstein used plain covariance instead

Chapter 3

Designing the attack

Before designing my own attack I tested Bernstein's original attack on my Intel Core i5-6300HQ machine running Linux 5.0. Based on my observations I then designed a modified attack better suiting its intended use.

3.1 Testing Bernstein's attack

Bernstein included the whole source of his attack in [4], allowing me to build it myself. I built the server against my system-wide OpenSSL version 1.1.1b, the latest stable release today. The simplest way to test it was to run both stages of the attack on the same machine.

Running the attack in its original form brought fruitless results, not detecting any of the key's bytes.

After a closer inspection of the client code, I found a hard-coded tail clip cutoff for timings. This value was apparently tuned for a Pentium III, defined at 10 000 ticks per single block encryption. Observing the data dumped by the client program, I noticed the threshold was almost 30x higher than the average encryption time. After tweaking the threshold to 5x the average, the attack cracked several bits of the key. These are shown in table 3.1 as the number of candidates for each byte of k.

Bernstein uses the x86 RDTSC instruction, which reads the timestamp counter register. Therefore context switches have a deeply negative impact on the precision of measurement.

i	0	1	2	3	4	5	6	7
candidates	32	80	256	256	113	69	256	256
i	8	9	10	11	12	13	14	15
candidates	241	96	256	256	228	94	256	256

Table 3.1: Number of candidates for bytes of k detected by Bernstein's attack

3.2 Defining improvements

The partial success of the modified Bernstein's software convinced me to build my work on top of it using the same core principle but bringing these modifications:

1. The network aspect including variable-size packets is dropped and all the work happens on a single machine
2. It compiles and runs under Linux and Microsoft Windows 10, considering both operating systems are installed on the machines in the InfoSec laboratory at FIT CTU
3. A variable number of "study" keys are observed and their timings correlated to the target key's dataset
4. The cutoff threshold for discarding excessive timings is determined at runtime to account for differently performant machines
5. The tested AES implementation is chosen by the user
6. Output data is visualized and sorted to facilitate manual analysis
7. Pearson's correlation coefficient is used to define correlations
8. Caches may be optionally purged before each encryption to measure their impact on the attack
9. Data visualizer is added for convenience
10. The attack is automated and configurable for various use cases

Chapter 4

Implementation

The final software is split into two units: an encryption core (further referred to as the “core”) written in C and an analytic wrapper (further on simply “wrapper”) written in Python. Additionally, a small 128-bit key generator written in Python is provided for convenience. This separation of tasks let me delegate visualization and data analysis to the robust Numpy and Matplotlib Python libraries while retaining maximum performance for the encryption itself.

The building of the core is handled by GNU Make on Linux and NMAKE (Microsoft Program Maintenance Utility) on Windows. The makefiles let the user configure options such as OpenSSL location or implementation selection, and provide targets for running tasks.

4.1 Encryption core

The core fulfills two roles: gathering correlation data and brute-forcing the key. The current role is chosen based on the presence (or lack thereof) of a brute-force candidate file, `bf.dat`.

4.1.1 Encrypting blocks

Data blocks to encrypt are generated pseudo-randomly. The user may choose between the standard C `rand`, or OpenSSL’s `RAND_bytes` PRNGs but `rand` seems to suffice and is faster. If encryption time exceeds the cutoff threshold, it is performed again until the criterion is met.

Testing of custom implementations is enabled by the supplied `aes.h` header which must first be implemented. This entails three functions for initialization, key expansion, and encryption. The custom implementation is then added to the makefile via a provided variable.

4.1.2 Gathering correlation data

The target key k may be provided in a file or randomly generated. The number N_k of test keys to correlate against k and the number N_b of n to encrypt and measure per key is chosen at compile time.

Before actual data gathering takes place N_b block encryptions are measured without a threshold and total time is measured using a monotonic timer to calculate the average encryption time. The cutoff threshold is then defined as a fixed multiple of this average and the rate of encryption per second is dumped for use by the wrapper.

N_b block encryptions are then measured for k and N_k randomly generated test keys. The correlation is calculated for each test key as described in section 2.2.2 and summed per k byte candidate. As Pearson's correlation coefficient returns values from $[-1, 1]$, the resulting correlation values for each k byte candidate lie in $[-N_k, N_k]$. These are sorted per byte and dumped to `corr.txt` for analysis by the wrapper.

4.1.3 Brute force attack

When a `bf.dat` file exists, the core tries to break k by brute force. The file contains candidates for each k byte sorted by relevance. It is assumed that the more candidates exist for a given byte, the less likely the correct value is at the front. Therefore the bytes with the least candidates are rotated last while trying. Every candidate key encrypts a zero block and the result is compared to the scrambled zero produced by k.

4.2 Analytic wrapper

One of this python script's roles is visualizing the correlations for all 256 values of the 16 bytes of k using matplotlib, producing 16 graphs as shown in Appendix C. The other is automating the process of data collection and analysis.

4.2.1 Peak detection

After invoking the core and reading its correlation outputs, the wrapper attempts to detect significant peaks in the correlation values per k byte. As testing has shown in section 5, the bytes of k which leak data sport significant visually discernible peaks.

First, the mean absolute deviation is calculated for the data trimmed off 32 highest and lowest values. For explanation on this number see section 5.0.3. Peak values are then defined as all values exceeding a fixed multiple of this deviation. The method was chosen based on the observation of peak sizes which rarely exceed 32 bytes. Naturally, this method is not exact and tends to produce occasional false positives depending on the dataset.

4.2.2 Byte pools

Peak values gathered as k byte candidates are called a “byte pool”. For reasons detailed in section 5.0.1, the wrapper generates new data in a loop, adding the received correlations to the summary candidate dataset C. Once new data is gathered by the core, the wrapper performs peak detection, and the resulting candidates are intersected with the existing candidates in C. This results in a gradual reduction of pool sizes as more data is acquired. If a pool shrinks to zero, it is removed. Moreover, this method provides self-healing of false positives as the pools containing them are likely to be removed eventually, hinting false positives to the user.

All byte candidates in C and the pools are kept sorted by their sum of measured correlations. The wrapper uses encryption rate provided by the core to estimate the worst time a brute-force attack would take. When the time drops below a set threshold, the wrapper dumps the sorted candidates to `bf.dat`. For bytes associated with a pool, only the pool members are dumped. The core is then invoked to try breaking the key.

4.2.3 Checkpoints

To facilitate analysis, the wrapper backs up the received correlations after each collection and creates a checkpoint by dumping the current pools and sorted C to disk. The dumped C can be analyzed manually or visualized directly using the wrapper. Both give the user a detailed overview of the attack’s progression.

If the checkpoints are present, the wrapper loads them automatically at the start, effectively making the attack interruptible.

Chapter 5

Testing

To document all possible factors affecting the attack, I used my own T-box optimized implementation, which had already proven itself somewhat vulnerable to Bernstein's original attack. Unless stated otherwise, the tests were run on an Intel Core i5-6300HQ machine running Linux and the used params are $N_k = 1$ and $N_b = 2^{23}$.

5.0.1 ASLR

ASLR (address space layout randomization) is a feature of modern operating systems, including Linux and Windows, where the address layout of a new process is randomized. This is done by adding random offsets to the binary's sections. Under Linux, tweaking ASLR is done by setting the `kernel.randomize_va_space` kernel parameter to 0, 1, or 2. According to [9], the setting of 2 only adds heap randomization to the presets of 1. 0 turns ASLR off completely.

Multiple runs of the attack using the same k would sometimes leak different bits of the key. A quick experiment with turning off ASLR entirely caused the attack to always leak information about the same bytes. T-boxes in both mine and OpenSSL's fallback implementation are global constants placed in the (read-only) data segment. There should thus be no difference between the ASLR settings of 1 and 2. Table 5.1 shows the impact of ASLR on the attack and confirms that assumption. It also shows that disabled ASLR limits the attack to the bytes leaking in a single core run, making repeated core runs pointless.

Attempting to raise N_k to yield more data from a single run proved fruitless. The individual key correlations apparently leak identical data. While this makes the correlation peaks more pronounced and easier to detect, it doesn't let one crack more bytes of k. Setting $N_k = 20$ and re-running the same benchmark yielded exactly the same results, only the peaks were significantly amplified.

Runs	ASLR 0	ASLR 1	ASLR 2
1	3	4	3
5	3	10	9
10	3	13	12

Table 5.1: Number of leaking bytes of k over time for $N_k = 1$

I_{max}	{0, 4, 8, 12}	{1, 5, 9, 13}	{2, 6, 10, 14}	{3, 7, 11, 15}	Other
Runs	16	13	11	10	0

Table 5.2: 50 runs tallied by their I_{max}

5.0.2 T-boxes and memory layout

Observing the behavior of both my T-box implementation and OpenSSL's fallback implementation (discussed in section 5.3), I noticed the leaked bytes followed a pattern. A vast majority of vulnerable bytes are offset from each other by a multiple of 4:

$$\begin{aligned} I_{max} &= \{i, i+4, i+8, i+12\}, i \in \{0,1,2,3\} \\ I_k &\subseteq I_{max} \end{aligned} \tag{5.1}$$

where I_k denotes the indices of all bytes of k which leak data in a given run. I_{max} is a bounding set for I_k.

The occurrence of measurements where I_k contains additional vulnerable byte indices is very rare and if it happens, the extra bytes tend to leak a low number of bits. I measured data for 50 random keys and categorized them by their bounding I_{max} in table 5.2 which shows this behavior.

5.0.2.1 Critical indices

It is obvious that the spacing of I_{max} is not random. The definition of a T-box round in fig. 1.1 shows that the T-box T_i is indexed with bytes from the i-th column of s, ie. every 4th byte of s. As the first round of AES initializes s to $n \oplus k$, by def. 5.1 all bytes $k[i], i \in I_k$ are used in indexing a single T-box in the first round.

In other words, the leaks seem to be linked to a single T-box which is time-sensitive to certain indices i_c.

Let's define the bytes of n for each vulnerable index $i \in I_k$ which take an unusually long time to encrypt as $n[i]_{max}$. Then

$$i_c = \{n_c[i] \oplus k[i] \mid n_c \in n[i]_{max}, i \in I_k\}$$

As all vulnerable bytes of s are used with the same T-box, the values of i_c should stay the same for all $i \in I_k$.

I_k	$k[i], i \in I_k$ (hex)	$n[i]_{max}, i \in I_k$ (hex)	i_c (hex)
{1, 9, 13}	{8e, a4, 20}	{{a2}, {88}, {0c}}	{2c}
{1, 9, 13}	{1e, 08, df}	{{32}, {24}, {f3}}	{2c}
{1, 9, 13}	{e7, da, bd}	{{cb}, {f6}, {91}}	{2c}
{1, 9, 13}	{5a, 77, 1e}	{{76}, {5b}, {32}}	{2c}

Table 5.3: Values of i_c taken for 4 random keys. T_0 base offset: $8f60_{16}$

This supports Bernstein's statement about time-dependent array indexing. I tried to detect i_c values with disabled ASLR to prevent the tables from changing addresses. The measurements in table 5.3 show that the values of i_c indeed remain constant across indices in I_k when using various keys k with a constant memory layout.

As i_c holds a single value of $2c_{16}$, the vulnerable bytes of k in the used configuration may be defined as

$$k[i] = 2c_{16} \oplus n[i]_{max}[0], i \in I_k$$

Similar results were obtained using OpenSSL's `aes_core.c` implementation discussed in section 5.3 but with a different (yet also single) value of i_c, likely caused by a different memory layout and implementation details. Enabling ASLR or otherwise changing the T-box addresses in memory had the expected effect of changing i_c and sometimes also changing I_k.

These observations showed that a single T-box is responsible for the timing leaks and that the critical indices i_c may be detected by testing with known keys.

5.0.3 Leaking bits

Most core runs do not leak all 8 bits of a vulnerable key byte. Instead, they usually leak at least 3 bits, yielding candidate pools capped at 32 bytes. This observation led me to design the 32-byte symmetric cutoff used in peak detection by the wrapper.

The bytes which leak less than 8 bits have a strong tendency to leak the highest bits first, creating a contiguous interval of candidates (see table 5.4 and fig. 5.2). This means the values of i_c also form a contiguous interval of indices of a T-box which is associated with high load times.

Graph note

Starting with fig. 5.1 I present graphs plotting the total correlation values for all possible values of a k byte. The red dotted lines denote the maximum, not the actual value of $k[i]$.

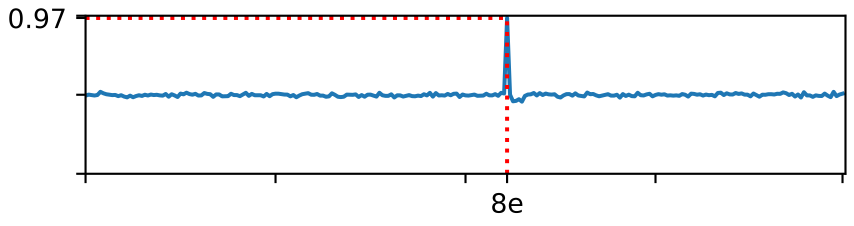

Figure 5.1: A full-byte leak of $k[1] = 8e_{16}$

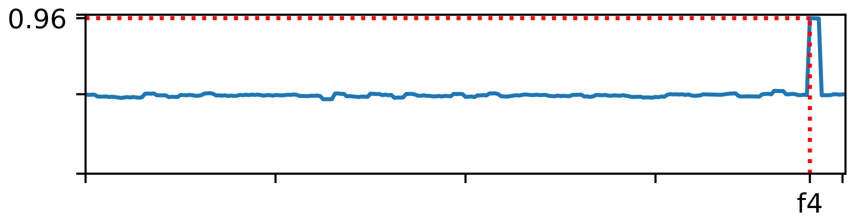

Figure 5.2: A 6-bit leak of $k[0] = f5_{16}$

$i \in I_k$	Candidates (hex)	Bits leaked
0	1d 1e 1c 1f	6
4	51 52 50 53	6
8	24 26 27 20 25 22 21 23	3
12	70	8

Table 5.4: Key candidates after a single core run

5.0.4 Connection with caches

The correlation between memory layout and leaking bytes suggests that cache penalties are causing the timing spikes. If a certain index of a T-box caused consistent cache misses, the resulting miss penalties could add up to a statistically significant timing spike.

To try and confirm the hypothesis, I tried purging all levels of cache before every encryption. This would ensure the first round couldn't make use of cached T-boxes.

Doing this didn't stop the attack but made it slower and more difficult. Figure 5.3 shows that secret data is still at least partially leaking.

OpenSSL's `aes_x86core.c` file contains a prefetch function for prefetching the last T-box (reading the whole T-box and caching it in the process). I tried using the same function in my implementation of T-box AES, but it didn't stop the attack no matter which T-boxes I tried to prefetch.

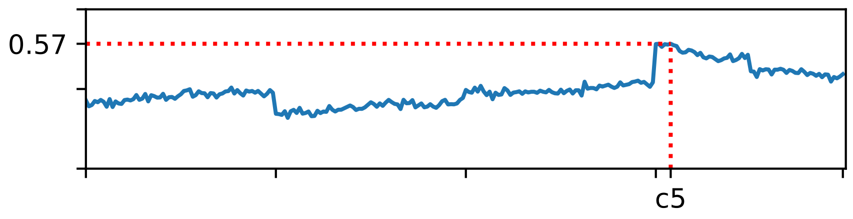

Figure 5.3: A 3-bit leak of $k[0] = \mathrm{c0}_{16}$ using purged caches

I was unable to get conclusive data on what exact role caches play in the attack, but it seems the role is supportive. This could be an interesting topic for future research.

5.1 The importance of cutoffs

Disabling the cutoff threshold for encryption time stops the attack. The same behavior made Bernstein's implementation non-functional when the threshold was set too high. Setting the cutoff too high or low makes the output very noise or very flat, respectively, both leading to failure. See Appendix C for an example with threshold cutoff turned off.

5.2 Testing custom implementations

I have three implementations I could test with the attack: a naive one, a T-box optimized one (the testing default), and one using Intel-specific AES-NI instructions [10]. The T-box implementation is vulnerable as it is the original target for this attack.

5.2.1 Testing a naive implementation

Since the naive implementation only uses a single 256-byte S-box table for lookup, there is a lower probability that it will leak timing data the same way. Running the test with default options showed no results, so I set the configuration to $N_k = 25$, $N_b = 2^{25}$ and still got no apparent leak patterns (fig. 5.4). The output is extremely noisy, and there are no outstanding peaks. I conclude the implementation as resistant, although such implementations are not usually found in production.

5.2.2 Testing an AES-NI implementation

These instructions are using hardware-assisted encryption, hence cache and RAM access times cannot help with the attack. Tweaking parameters to

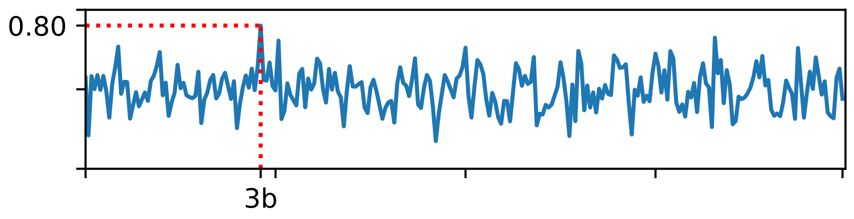

Figure 5.4: Naive implementation is resistant ($k[0] = 4f_{16}$)

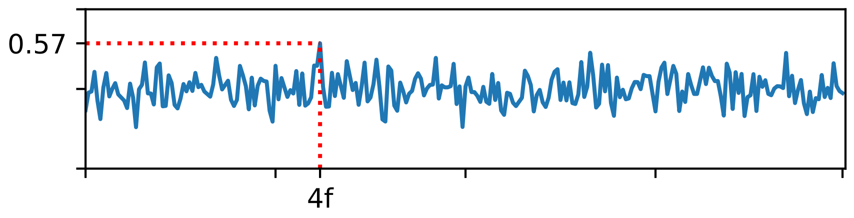

Figure 5.5: AES-NI implementation is resistant ($k[0] = 96_{16}$)

$N_k = 10, N_b = 2^{30}$ didn't help (fig. 5.5). The implementation is resistant.

5.3 Testing OpenSSL implementations

OpenSSL compiles with hardware encryption support enabled by default. The project includes assembly definitions for various CPU architectures including AES-NI support. Testing my system build of OpenSSL as well as OpenSSL built from source using the default configuration, I found no leaking bits, yielding noisy and empty results similar to fig. 5.5. I tested the default build on two machines: the default Core i5-6300HQ machine which supports AES-NI and a Core 2 Duo powered Linux machine which does not but it supports SSE3 instructions. Neither of them leaked secret data.

OpenSSL does have a software implementation in the `aes_core.c` file which is T-box optimized. This implementation is only used as a fallback when the CPU does not support any of the more specific implementations or when the *no-asm* option is used. This explains why the default build on the Core 2 Duo machine stayed resistant despite lacking AES-NI support. It used an x86-specific constant-time SSE3 implementation instead.

Today finding a machine incapable of using any of the provided specific implementations is difficult. Therefore I compiled OpenSSL with the *no-asm* option and found it vulnerable similar to my T-box implementation (fig. 5.6). Very similar results were obtained using Windows 10 on an i5-8600K machine.

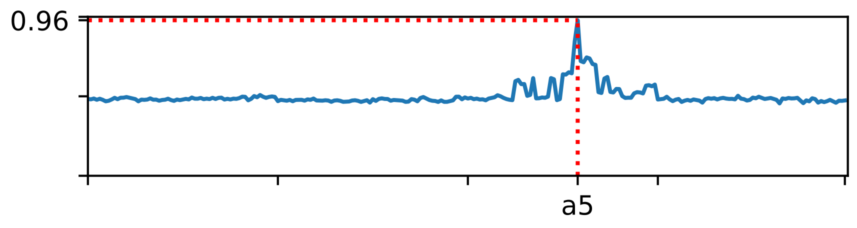

Figure 5.6: OpenSSL *no-asm* implementation is vulnerable ($k[8] = \text{a5}_{16}$)

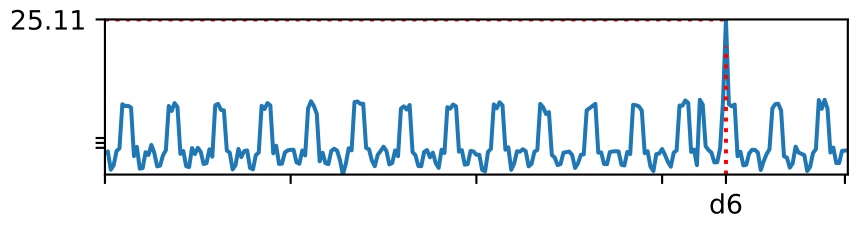

Figure 5.7: OpenSSL *no-asm* implementation on Core 2 Duo ($k[11] = \text{d6}_{16}$)

Testing the *no-asm* build on the Core 2 Duo (fig. 5.7) yielded some very interesting results where multiple bytes leaked 64 candidates regularly split into quartets. This hints at multiple evenly spaced critical indices, possibly due to the Core 2 Duo's smaller caches. See Appendix C for an additional measurement showing this behavior.

5.4 Testing on Windows and other machines

Using the attack on Windows is possible. See Appendix C for an example of a moderately leaked key. However, I noticed that the attack is weaker on Windows, revealing data more slowly and having more false positives in the pools. Nevertheless, the attack still works and leaks secret data.

Chapter 6

Laboratory assignment

The intended use of this software is a laboratory assignment at the hardware security course at FIT CTU. The course leads students to develop several implementations of AES-128 of ever-increasing performance.

I prepared guidelines for a possible assignment that would lead the students through understanding this attack's principle, testing their implementations and possibly testing OpenSSL on their machines. The guidelines along with a guide to building and using the software are present in `README.md`. The attack is hardware and OS dependent, so students will get the chance to compare the attack's performance on various machines, including the provided lab PCs.

The first step is to introduce the students to the concept of timing attacks in general, and this specific attack. To achieve this, I removed three topical portions of the core's code, prompting students to fill them in. The portions are: tallying encryption times, calculating mean encryption times per each key byte, and correlating two datasets as detailed in section 4.1.2.

The second step involves implementing the `aes.h` header for all of the student's currently finished implementations of AES-128 and testing them for vulnerability. This should give the students a comparison between the hardware-accelerated and immune AES-NI implementation and the software-optimized yet vulnerable T-box implementation.

The final step may consist of testing the attack on OpenSSL and comparing the vulnerability of its fallback implementation to the students' own. This step is mostly informative but could be interesting for curious students.

In the end, more parts of the core may become fill-ins and the final form of the laboratory exercise will be subject to personal discussions with the course's teachers.

Conclusion

The main goal of the thesis was to demonstrate the feasibility of a timing side-channel attack on the AES-128 cipher using modern processors, and it was successfully fulfilled. I expanded on an already existing attack and produced software exploiting a timing side-channel vulnerability in a provided AES-128 implementation. The software is, in some cases, capable of extracting the whole 128-bit key.

Tests were made on three modern x86 processors with L3 caches: Intel Core i5-6300HQ, Intel Core i5-8600K, and Intel Xeon E3-1245 v6 using both Linux and Windows 10 OS. The main results of my findings are as follows:

- A naive implementation following the FIPS description is resistant
- A Custom T-box optimized implementation is vulnerable
- OpenSSL 1.1.1b `AES_encrypt` compiled with the *no-asm* option is also vulnerable
- OpenSSL 1.1.1b `AES_encrypt` using the default build options is resistant
- Vulnerable implementations may be attacked multiple times and leak additional bits as long as ASLR is enabled
- The above may, in many cases, be applied to the point where a brute-force attack is feasible within a reasonable time

Additional findings include documenting the effect of other factors on the attack, such as the used OS, prefetching lookup tables or filtering gathered timing data. I analyzed the patterns in leaking key bytes and presented my explanations for them.

The software is presented in a form suitable for laboratory assignments, including the necessary documentation and proposed tasks for the students. It has been tested in the target environment at FIT CTU and is ready to be used in education.

Bibliography

1. BOGDANOV, Andrey et al. Biclique cryptanalysis of the full AES. In: *International Conference on the Theory and Application of Cryptology and Information Security*. 2011, pp. 344–371. Available from DOI: `10.1007/978-3-642-25385-0_19`.
2. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Advanced Encryption Standard (AES): Federal Information Processing Standards Publication (FIPS) publication 197. 2001. Available from DOI: `10.6028/NIST.FIPS.197`.
3. AUTI, Harshada M. et al. Implementation of AES Encryption and Decryption using T-Boxes on FPGA. *International Journal of Computer Technology and Electronics Engineering (IJCTEE)* [online]. 2014, vol. 4, no. 3 [visited on 2019-04-21]. ISSN 2249-6343. Available from: `http://www.ijctee.org/files/VOLUME4ISSUE3/IJCTEE_0314_04.pdf`.
4. BERNSTEIN, Daniel J. Cache-timing attacks on AES [online]. 2005 [visited on 2019-04-21]. Available from: `http://palms.ee.princeton.edu/system/files/Cache-timing+attacks+on+AES.pdf`.
5. KOCHER, Paul C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems [online]. 1996 [visited on 2019-05-12]. Available from: `https://www.paulkocher.com/doc/TimingAttacks.pdf`.
6. WEI, Liu et al. Cache-timing Attacks on AES [online]. 2013 [visited on 2019-04-21]. Available from: `http://sites.nyuad.nyu.edu/moma/pdfs/moma-project-cache-timing-attacks-on-aes.pdf`.
7. ASHOKKUMAR, C et al. “S-Box” Implementation of AES is NOT side channel resistant. *IACR Cryptology ePrint Archive* [online]. 2018 [visited on 2019-04-21]. Available from: `https://eprint.iacr.org/2018/1002.pdf`.

8. ZHANG, Liwei et al. Statistical Analysis for Access-Driven Cache Attacks Against AES. *IACR Cryptology ePrint Archive* [online]. 2016 [visited on 2019-04-21]. Available from: `https://eprint.iacr.org/2016/970.pdf`.

9. *Documentation for /proc/sys/kernel/** [online] [visited on 2019-05-13]. Available from: `https://www.kernel.org/doc/Documentation/sysctl/kernel.txt`.

10. *Intel® Advanced Encryption Standard Instructions (AES-NI)* [online] [visited on 2019-05-15]. Available from: `https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni`.

Appendix A

Acronyms

AES	Advanced encryption standard
PC	personal computer
OS	operating system
CPU	central processing unit
NIST	U.S. National Institute of Standards and Technology
PRNG	pseudo-random number generator
SSE3	Streaming SIMD Extensions 3
InfoSec	information security

Appendix B

Contents of enclosed CD

```
README.md ................... markdown document with usage instructions
src ........................................ the directory of source codes
  aes_tbox.c .................... a vulnerable AES-128 implementation
text ............................................ the thesis text directory
  latex ................ the directory of LaTeX source codes of the thesis
  thesis.pdf ........................... the thesis text in PDF format
```

Appendix C

Measurements

Examples of full 16-byte graphs from various measurements. Every measurement has the following attributes: N_k (number of used keys), N_b (number of encryptions per key), N_c (number of core runs), implementation, CPU and OS.

The key k = `99:7c:d2:3c:94:da:ae:85:dd:2d:68:d6:bb:88:4f:b0` is used for all measurements taken in this section. Each graph represents correlation sums over N_c core runs for all possible values of $k[i], i \in [0, 15]$. The red lines indicate maxima for each byte's dataset. Read the graphs left to right, top to bottom (the top left graph indicates $k[0]$, the bottom right $k[15]$).

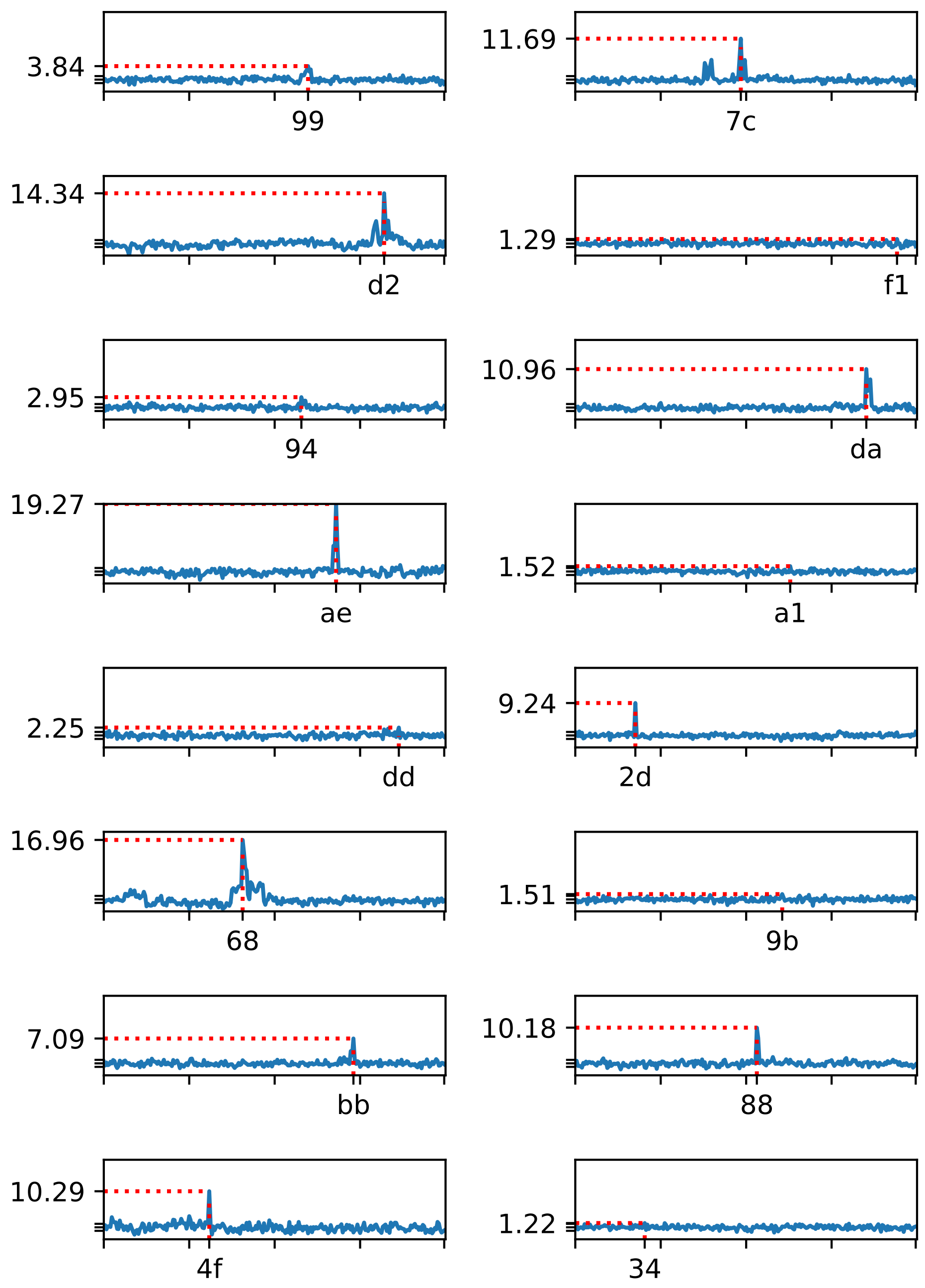

Figure C.1: Testing in the InfoSec lab at FIT CTU. A brute-force attack would break the rest of k in less than a day.
CPU: Intel Xeon E3-1245 v6, OS: Windows 10, Impl.: custom T-box, $N_k = 10$, $N_b = 2^{23}$, $N_c = 15$

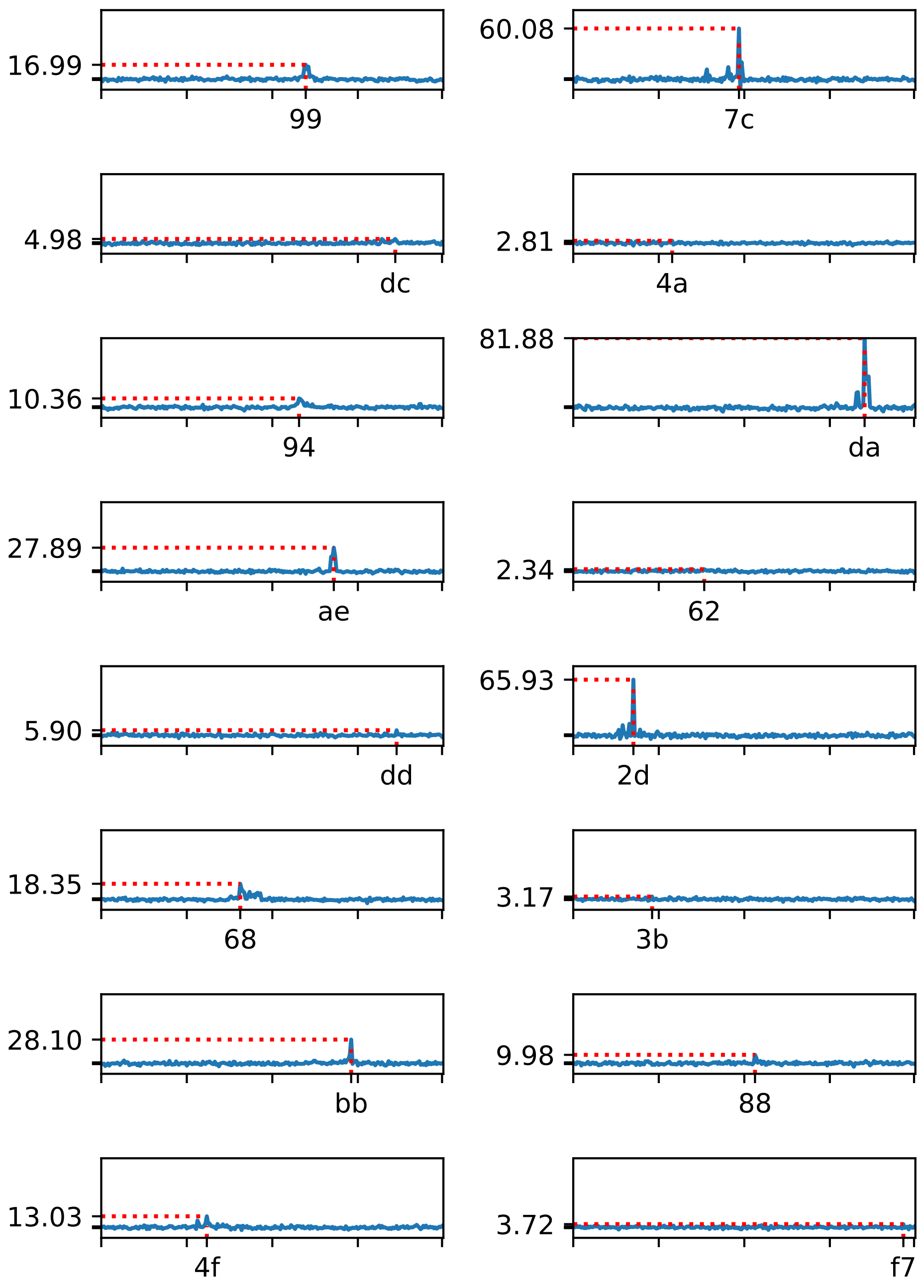

Figure C.2: Brute-force attack took 2 seconds after 25 runs of the core CPU: Intel Core i5-8600K, OS: Windows 10, Impl.: custom T-box, $N_k = 10$, $N_b = 2^{23}$, $N_c = 25$

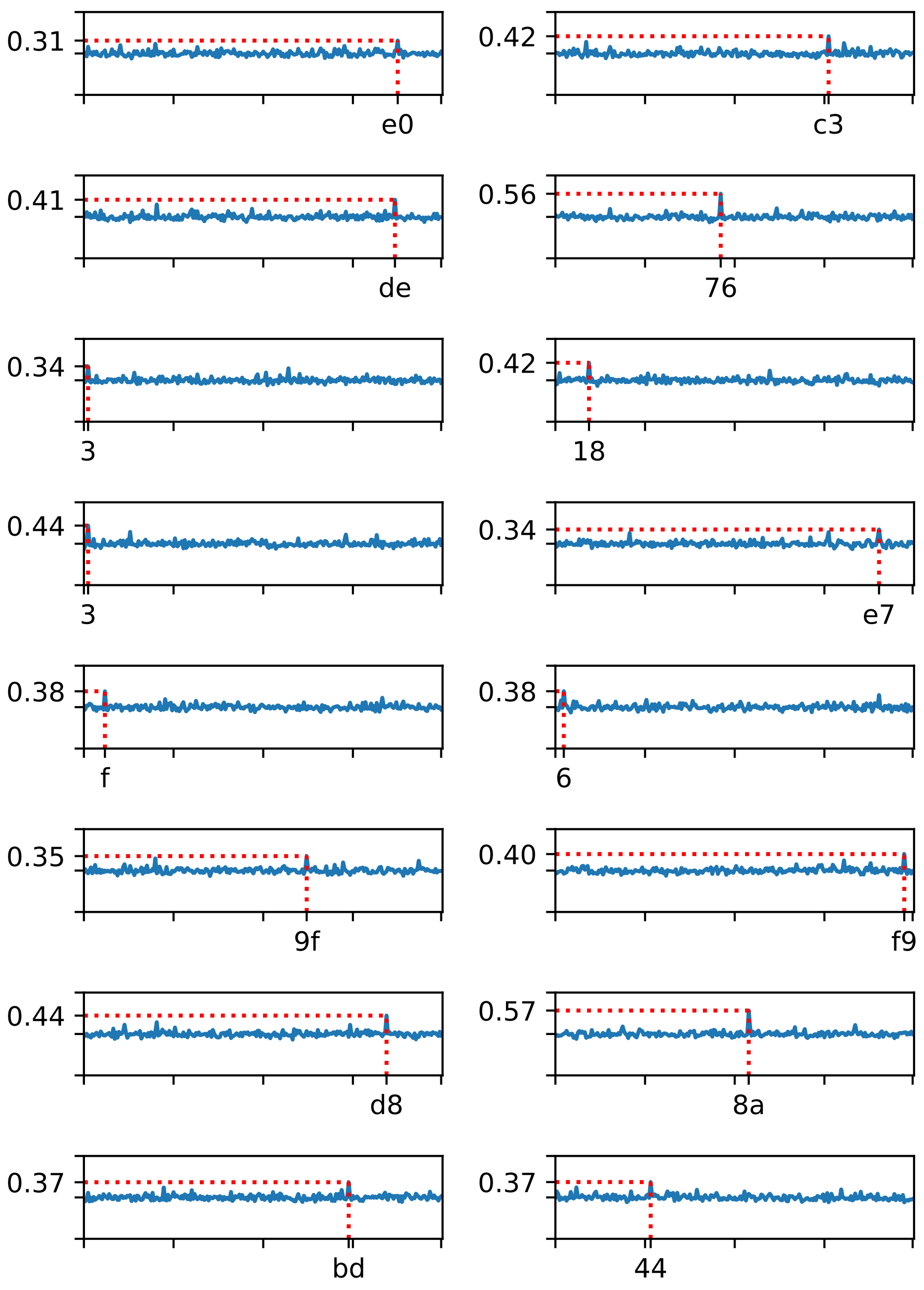

Figure C.3: Removing the cutoff yields noisy and bad data (notice the misleading spikes).
CPU: Intel Core i5-6300HQ, OS: Linux, Impl.: custom T-box, $N_k = 10$, $N_b = 2^{23}$, $N_c = 1$

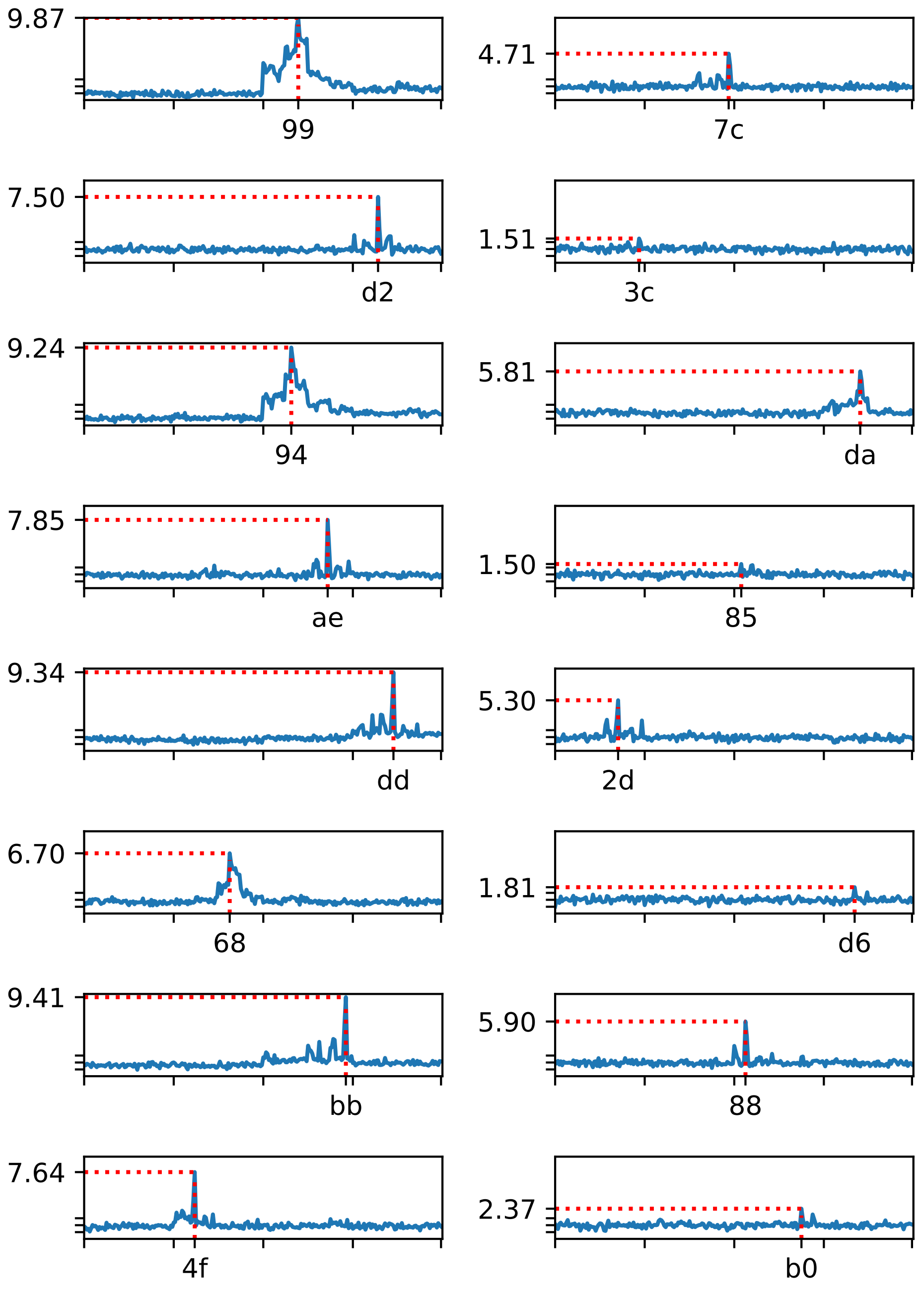

Figure C.4: After 25 runs of the core, a brute-force attack would take less than a second.
CPU: Intel Core i5-6300HQ, OS: Linux, Impl.: OpenSSL no-asm, $N_k = 1$, $N_b = 2^{23}$, $N_c = 24$

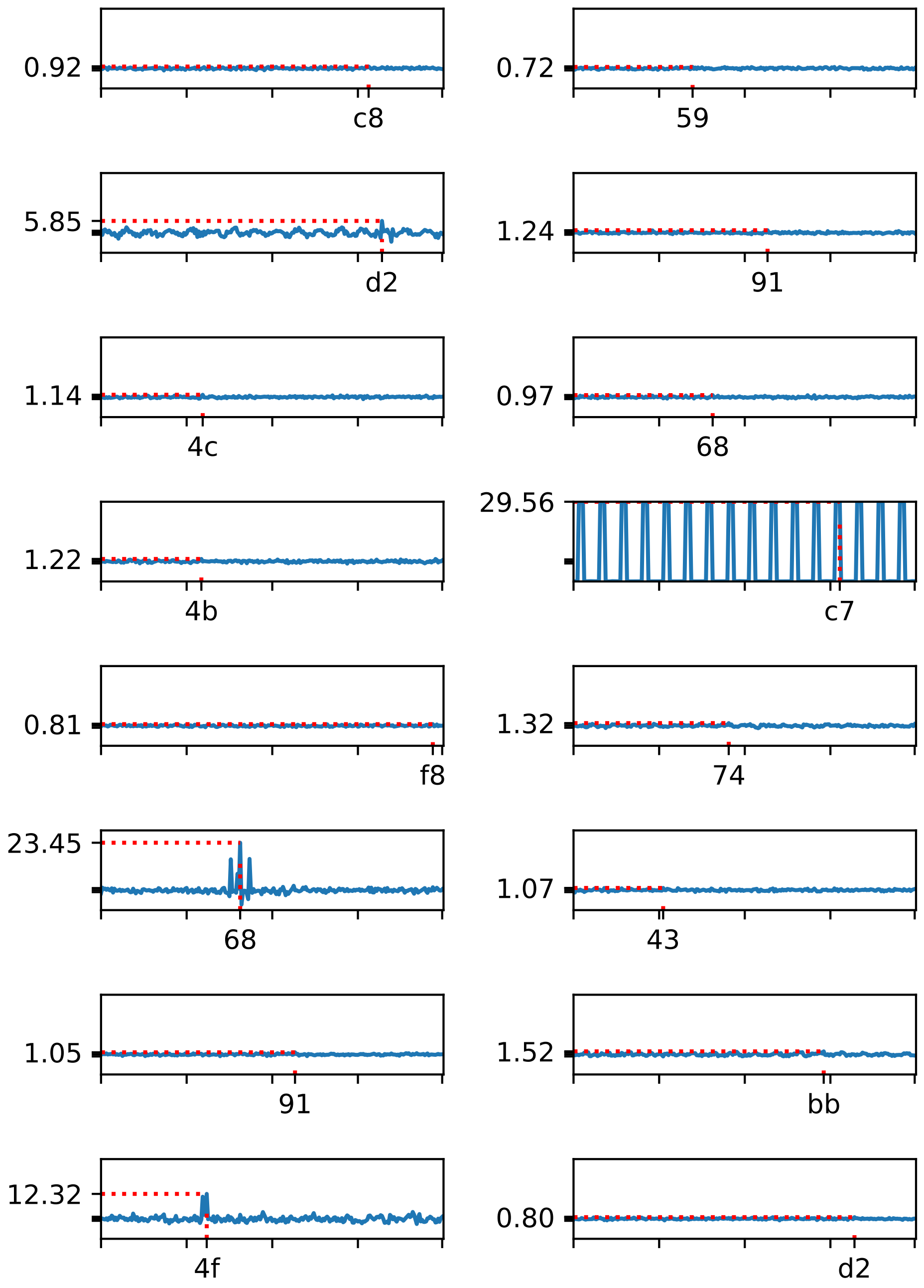

Figure C.5: The wave pattern ($k[7]$) appears even with my T-box implementation.
CPU: Intel Core 2 Duo, OS: Linux, Impl.: custom T-box, $N_k = 5$, $N_b = 2^{22}$, $N_c = 5$