Automatic Detection and Decryption of AES by Monitoring S-Box Access

Josef Kokeš, Jonatan Matějka and Róbert Lórencz
Department of Information Security, Faculty of Information Technology, Czech Technical University in Prague, Thakurova 9, Praha 6, Czech Republic

Keywords: AES, Rijndael, Cipher, Encryption, S-Box, Key Recovery, Plaintext Recovery, Dynamic Analysis.

Abstract: In this paper we propose an algorithm that can automatically detect the use of AES and automatically recover both the encryption key and the plaintext. It makes use of the fact that we can monitor accesses to the AES S-Box and deduce the desired data from these accesses; the approach is suitable to software-based AES implementations, both naïve and optimized. To demonstrate the feasibility of this approach we designed a tool which implements the algorithm for Microsoft Windows running on the Intel x86 architecture. The tool has been successfully tested against a set of applications using different cryptographic libraries and common user applications.

1 INTRODUCTION

In modern IT, security is no longer considered an afterthought; quite the contrary, security is a mandatory feature of many hardware and software products. In recent months we have seen a major push for increasing security which manifested e.g. in the announcements by browser manufacturers that they plan to abolish or at least minimize unsecured web traffic in near future (DeBlasio, 2020), or in the deprecation of TLS protocols versions 1.0 and 1.1 from all major browsers and other applications in early 2020. At the same time, privacy concerns of the users are on the rise (Auxier et al., 2019).

While the improvements in security often improve privacy as well, this is not always the case. In particular, the increased use of encryption to protect communication from outsiders can also remove control of the legitimate users over the transmitted data as they are no longer able to easily monitor the contents of the network traffic. This is especially dangerous in case of applications which are considered legitimate by users but do not provide any means of verification what is being sent over the network. For example, an operating system or an application may very well propose to send telemetry data to the developer to improve the user experience, but if the source code of that software is not available, the user cannot easily verify what data is actually being collected. The user could, of course, resort to the techniques of reverse engineering, but that almost always requires so much effort as to make this approach prohibitively expensive.

In this paper, we propose an alternative solution to this problem: We demonstrate an algorithm which can automatically detect encryption and decryption with the AES cipher and automatically recover both the encryption keys and the plaintext data. We achieve that by modifying the control flow of the target application to monitor accesses to the substitution tables used for the operation of the cipher and then deducing the actual plaintext data from these accesses.

2 PRELIMINARIES

In this section we will briefly introduce the common ways of implementing AES on Intel-based architectures.

2.1 Common Implementations of AES

AES is probably the most commonly used block cipher in the world. It's design and structure is defined in (Daemen and Rijmen, 2002) and can theoretically be implemented according to that as well. It is, however, common to tailor the implementation to the specific features in the target CPU. Note that since the key-expansion process is generally a one-time operation or at least is performed much more rarely than the actual encryption, it is usually not optimized on the algorithmic level and if any optimization is used

Kokeš, J., Matějka, J. and Lórencz, R.
Automatic Detection and Decryption of AES by Monitoring S-Box Access.
DOI: 10.5220/0010255201720180
In *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP 2021)*, pages 172-180
ISBN: 978-989-758-491-6

at all, it's left to the compiler and its choice of instructions and their ordering. For that reason, we will only discuss the operations used in the actual encryption/decryption.

When targeting the Intel architectures (IA-32, Intel 64), the following approaches are usually taken:

2.1.1 The Naïve Implementation

The naïve implementation of AES follows the operations described in the cipher's specification, i.e. key-expansion, sub-bytes, shift-rows, mix-columns and add-round-key, in the designated order and the specified number of repetitions.

The sub-bytes operation is commonly implemented through a lookup table of 256 values where the input to sub-bytes is used as an index to the table and the output value is read from that location in the table; another such table is used for the inverse of sub-bytes. That removes the need for calculating inverses in AES's Galois field.

Shift-rows is typically implemented as described as reordering of the bytes in the encryption state.

Mix-columns may either be implemented as straight table multiplication or optimized by pre-calculating the necessary multiples for each possible input value. For encryption, we need multiples of two and three, for decryption multiples of 9, 11, 13 and 14. That can be done by using 6 pre-calculated tables with the same structure.

Add-round-key is again usually implemented in a straightforward XOR, although multiple bytes may be processed at once using e.g. 32-bit XOR instructions.

An example of this optimized approach can be found in (Malbrain, 2007).

2.1.2 Implementation using T-tables

Assuming that the target CPU architecture supports 32-bit instructions, further pre-calculation allows us to optimize the encryption process to just four lookups and four XOR operations per column per round, as described in (Daemen and Rijmen, 2002):

Given an input state of $A = [a_{i,j}], 0 \leq i < 4, 0 \leq j < N_b$, expanded round key $K = [k_{i,j}], 0 \leq i < 4, 0 \leq j < N_b$ where N_b is the number of columns of A, we can express the encrypted output state $D = [d_{i,j}], 0 \leq i < 4, 0 \leq j < N_b$ as:

$$\begin{pmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{pmatrix} = \sum_{i=0}^{3} T_i(a_{i,j+C_i}) + \begin{pmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{pmatrix}, \qquad (1)$$

where $+$ is the operation XOR, C_i are the respective left shifts for the i-th row of the state (e.g. 0, 1, 2 and 3 respectively for the 128-bit key version of AES) and T_i are pre-calculated as:

$$\begin{aligned} T_0(x) &= \begin{pmatrix} 2 \cdot S[x] \\ 1 \cdot S[x] \\ 1 \cdot S[x] \\ 3 \cdot S[x] \end{pmatrix}, T_1(x) = \begin{pmatrix} 3 \cdot S[x] \\ 2 \cdot S[x] \\ 1 \cdot S[x] \\ 1 \cdot S[x] \end{pmatrix}, \\ T_2(x) &= \begin{pmatrix} 1 \cdot S[x] \\ 3 \cdot S[x] \\ 2 \cdot S[x] \\ 1 \cdot S[x] \end{pmatrix}, T_3(x) = \begin{pmatrix} 1 \cdot S[x] \\ 1 \cdot S[x] \\ 3 \cdot S[x] \\ 2 \cdot S[x] \end{pmatrix} \end{aligned} \qquad (2)$$

for all possible byte values of x, given that $S[x]$ is the result of the sub-bytes transformation of x.

The T-tables can be further compressed by observing that they are in fact rotated versions of each other and as a result only one of them needs to be pre-calculated, the others can be obtained from it through the use of rotation.

Further compression is possible if an unaligned data access is possible because then the tables can be stored overlapped.(Rijmen et al., 2000)

2.1.3 Implementation using Bit-slicing

The bit-slicing implementation (Käsper and Schwabe, 2009) is inspired by the hardware-based implementations of AES: The cipher's state is represented as a series of bits and the operations usually performed by hardware gates are simulated using logical operations. This approach has several significant advantages: it does not need any pre-calculated tables (the lookups are replaced by series of logical operations), reducing memory requirements of the algorithm, the algorithm takes a constant time in clock cycles as the operation sequences are fixed regardless of any variations in input data, and timing attacks on memory access are difficult if not impossible (Matsui, 2006). The chief disadvantage is the reduced speed of the algorithm, although that can be offset if vector instructions (such as those provided by the MMX, SSE, SSE2 etc. instruction sets) are used and we can process multiple blocks of the cipher in parallel, e.g. in the CTR encryption mode: we would "slice" bits from eight independent states into eight 128-bit registers and process them in parallel.

2.1.4 Implementation using AES-NI

In 2008, Intel introduced a new instruction set extension for a hardware support of AES encryption and decryption (Gueron, 2010). It consists of 6 instruction which provide encryption and decryption of a single round of AES as well as support for key expansion and the inverse Mix-columns operation. Much

like bit-slicing, this technique provides high security (e.g. resistance to timing attacks) and low memory footprint because it does not use any in-memory tables, and in compared to bit-slicing provides a very high performance due to its hardware-based implementation. Another benefit is the very simple implementation, although care needs to be taken to verify that the AES-NI instruction set is actually available at the CPU where the code is running – customarily, the code would check for the presence of these instructions and then branch to either an AES-NI based version or a traditional version based on one of the approaches shown above.

3 OUR APPROACH

Our goal is to detect the use of AES automatically and also automatically recover encryption keys and plaintexts. We use dynamic analysis to achieve it – we attach to the analyzed application as a debugger and then make use of the debugging APIs to monitor data accesses to the precalculated tables and deduce both the key and data from them. Obviously, this approach is only applicable to AES implementations which make use of these tables, i.e. the naïve implementation or the T-tables implementation.

3.1 Locating Tables

In order to be able to monitor accesses to the tables, we need to locate them in the target application's memory first. To do that, we use `VirtualQueryEx` function to get the list of memory pages belonging to the analyzed process, copy these pages to our memory using `ReadProcessMemory` and then search them. Since the tables may be stored in a variety of fashion, we do not compare memory blocks to known values but rather study the relationships between bytes – we are looking for multiplications of the original SubByte table[1] with common interleaving (1 byte for SubBytes, 4 bytes for T-Tables and 8 bytes for overlapping T-Tables).

With many applications, it is sufficient to perform the search only once at the beginning of the application because the tables are statically compiled into the application. Some applications, however, build these tables at least partially dynamically during their runtime – e.g. to calculate the T-tables from the statically stored SubBytes table or to load a dynamic library which contains these tables. To facilitate support for these applications, we perform the search repeatedly using a background thread; currently no performance optimizations are performed for this search, but it seems likely that some would be applicable.

[1] 1, 2 and 3 for encryption tables and 1, 9, 11, 13 and 14 for decryption tables

3.2 Monitoring Access

Once we have located the substitution tables, we need to monitor access to them. Based on the specific hardware used, there may be different ways of doing so. On the Intel architecture, we could use debug registers (Intel Corporation, 2019) for this purpose, but unfortunately only for memory locations of up to 8 bytes each could be monitored, which is not enough to detect all accesses – even SubBytes is at least 256 bytes long, T-tables even longer.

Instead, we decided to make use the concept of memory paging and memory page protection: Once we know in which memory pages the substitution tables reside, we remove all access from these pages using `VirtualProtectEx` by adding the `PAGE_GUARD` flag. When that was done, any access to any location within the memory page causes a page fault exception before passing it to the application itself the active debugger – our tool – is notified about it through a debug event. Specifically, we learn of the actual memory location and the type of access (read, write, execute) that caused the fault. We can then verify whether the access was a substitution table access and if so, process it accordingly.

Obviously, we must allow the application to actually perform the table access so that it can continue in its execution. We achieve that by temporarily removing the `PAGE_GUARD` flag from the affected memory page, enabling the single-step (trap) flag in the thread's `FLAGS` register and resuming the thread; after a single instruction the single-step flag causes another debugging event which we capture and restore both the memory protection by adding the `PAGE_GUARD` flag to the memory page and the standard thread execution by clearing the single-step flag from `FLAGS`.

3.2.1 Special Considerations

While the monitoring process is fairly straightforward, care needs to be taken to facilitate several special situations.

In particular, we need to consider the possibility that the substitution tables are stored in the code segment rather than data segment, such as in (Polyakov, 2016). In that case an attempt to execute an instruction from the same memory page will cause a page fault because the instruction itself cannot be read due to the protection settings. That can be solved by

checking whether the access occurred inside a substitution table or whether it occurred somewhere else within the monitored memory page, and in such a case simply removing the protections, single-stepping the instruction and then restoring the protections. It will degrade the performance significantly but the code will function as expected.

Unfortunately, that is not the case if the instruction that accesses the substitution table is located within the same memory page as the substitution table itself: In this case, we would fail to detect the table access because we removed the protection in order to execute the instruction and will only restore it after the instruction has completed, i.e. after the table access. We solve this problem by decoding the instruction in software using a third-party library and determining whether it is this particular case; if it is, we emulate the instruction rather than execute it directly.

3.3 Monitoring Key Expansion

During key expansion, the substitution tables are used to perform a SubWord operation:

$$W_i = \begin{cases} K_i & \text{for } i < N_k \\ W_{i-N_k} + r(s(W_{i-1})) + rcon_{i/N_k} & \text{for } i \geq N_k, i \equiv 0 \pmod{N_k} \\ W_{i-N_k} + s(W_{i-1}) & \text{for } i \geq N_k, N_k > 6, i \equiv 4 \pmod{N_k} \\ W_{i-N_k} + W_{i-1} & \text{otherwise} \end{cases} \tag{3}$$

Here, K_i is the i-th column of the master key, N_k is the number of columns of the master key, W_i is the i-th column of the expanded key and functions r and w as well as the constant $rcon$ are defined as follows:

$$\begin{aligned} r(w) &= r\left(\begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix}\right) = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_0 \end{pmatrix} \\ s(w) &= s\left(\begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix}\right) = \begin{pmatrix} S[w_0] \\ S[w_1] \\ S[w_2] \\ S[w_3] \end{pmatrix} \\ rcon_i &= \begin{pmatrix} 2^{i-1} \\ 0 \\ 0 \\ 0 \end{pmatrix}, \text{for } i \in \mathbb{N}^+ \end{aligned} \tag{4}$$

In order to properly recover the key, we need to make several assumptions:

- While key expansion is being performed, no other access to substitution tables is performed except through the SubWord function.
- SubWord calls are performed in order of the columns in the expanded key.
- Accesses to the substitution tables are the same in all SubWord calls.

On the other hand, we do not make any assumption on the order in which the bytes in a word are being substituted – that might be influenced e.g. by aggressive optimizations on the part of the compiler while building the target application. We can, however, determine the proper ordering by verifying that the dependencies between columns do exist as expected.

The dependencies must be calculated separately for each size of the key. For example, with AES-128 we can make use of columns $\{x \mid x = 3+4k, k \in \mathbb{N}\}$ of the expanded key which depend on previous columns as shown in Figure 1. Then:

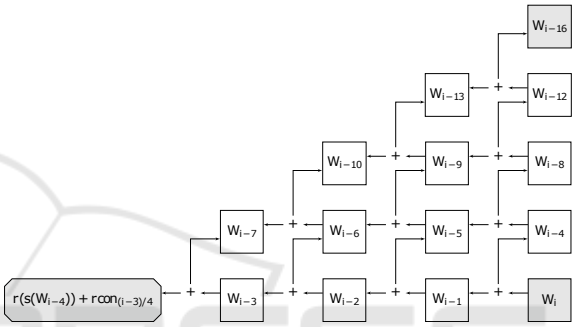

Figure 1: Dependency of columns in the expanded 128-bit key. The grayed boxes describe the final expression.

$$\begin{aligned} W_i =& W_{i-4} + W_{i-1} = \\ =& W_{i-8} + W_{i-5} + W_{i-5} + W_{i-2} = \\ =& W_{i-8} + W_{i-2} = \\ =& W_{i-12} + W_{i-9} + W_{i-6} + W_{i-3} = \\ =& W_{i-16} + W_{i-13} + W_{i-13} + W_{i-10} + W_{i-10} \\ &+ W_{i-7} + W_{i-7} + r(s(W_{i-4})) + rcon_{(i-3)/4} = \\ =& W_{i-16} + r(s(W_{i-4})) + rcon_{(i-3)/4} \end{aligned} \tag{5}$$

To detect these dependencies we then require five successive key columns. Since AES-128 performs 10 SubWord operations, we can produce six equations which describe the dependencies between all columns:

$$\begin{aligned} W_{19} &= W_3 + r(s(W_{15})) + rcon_4 \\ &\vdots \\ W_{39} &= W_{23} + r(s(W_{35})) + rcon_9 \end{aligned} \tag{6}$$

If the captured table accesses do not adhere to these expressions, then we know that the accesses are not a part of the key expansion process or the assumptions above have been violated. We can make use of this fact by finding the correct ordering of bytes

in each word by simply trying them all and checking which leads to satisfying all the expressions.

In this fashion we can can recover every fourth column of the expanded key $W_3, W_7, ..., W_{35}$. Then we can make use of the algorithm of key expansion to calculate the remaining columns, e.g. $W_{i+3} = W_i + W_{i+4}$ for $i \in 3, 7, ..., 35$, $W_i = W_{i+4} + r(s(W_{i+3})) + rcon_{(i+4)/4}$ for $i \in 0, 4, 8$ etc., eventually recovering all the columns of the key.

The key for AES-192 can be recovered in a similar fashion, although only two equations can be used to verify that we are indeed performing key expansion:

$$\begin{aligned} W_{41} &= W_5 + W_{17} + W_{29} + r(s(W_{35})) + rcon_6 \\ W_{47} &= W_{11} + W_{23} + W_{35} + r(s(W_{41})) + rcon_7 \end{aligned} \tag{7}$$

With AES-256, the recovery is complicated by the fact that not all columns which entered SubWord can be used for the expression of dependencies – we know the value of W_{27} and W_{31}, but we can not express it using other columns:

$$\begin{aligned} W_{39} &= W_7 + s(W_{35}) \\ W_{47} &= W_{15} + s(W_{43}) \\ W_{55} &= W_{23} + s(W_{51}) \\ W_{43} &= W_{11} + r(s(W_{39})) + rcon_5 \\ W_{51} &= W_{19} + r(s(W_{47})) + rcon_6 \end{aligned} \tag{8}$$

As a result, we do not have sufficient information to calculate the correct ordering of the bytes in a word; W_{27} and W_{31} allow for $4! = 24$ different valid orderings each, giving us $24^2 = 576$ different keys which all satisfy the defined expressions. This obstacle can be overcome by deferring the final calculation of the ordering until the encryption phase, behavior of which will help us detect which specific key was actually used.

3.4 Monitoring Encryption

During encryption we will observe substitution table access in every round of the cipher. Assume the following:

- While encryption is being performed, no other access to substitution tables is performed except by that block's encryption.
- All substitution table accesses are ordered exactly as the rounds themselves.
- No two rounds overlap.
- The data is being encrypted with the key expanded in the last monitored Key Expansion phase.

We do not make any assumptions on the order of accesses within one round.

The first input to SubBytes within a round is created simply as a sum of the plaintext and the first round key. It is passed through SubBytes and the output is then processed according to the cipher's specifications (rows shifted, columns mixed, next round key added) and forms the input to the second round's SubBytes. Repeat the process for the rest of the rounds, skipping MixColumns in the last round.

With the assumptions above, we know the expanded key, except possibly the ordering in some of its columns. We can make use of this information to determine the proper ordering of the states. Given S the input state of one round's SubBytes, T the output state of the previous round's SubBytes and K the round key, we know that:

$$S = \text{MixColumns}(\text{ShiftRows}(T)) + K \tag{9}$$

Unfortunately, we do not know the ordering of SubBytes calls for the individual bytes of states S and T. We can, however, re-formulate and relax the expression as:

$$\forall x : x \in \text{InvMixColumns}(S + K) \iff x \in T \tag{10}$$

We could now check all 16! possible permutations of state S and locate the matching one, but that would require quite a lot of computational power. We can, however, further relax the expression and apply it to each column of the state separately:

$$\forall x : x \in \text{InvMixColumns}(S_i + K_i) \implies x \in T \tag{11}$$

Now we only need to check $4 \times \frac{16!}{(16-4)!}$ variations of the ordering of S. For each selection we verify that all of its bytes appear in T. If that is not the case, then we know that we are not using the correct key. Otherwise we can apply the same reasoning to the next (or previous) round and express the condition on the whole sequence. E.g. if S was the SubBytes output of the last round and T its input, we can now focus on U the output of the second-to-last round's SubBytes, T its input and L its key. Then:

$$\forall x : x \in \text{InvMixColumns}(U_i + L_i) \implies x \in V \tag{12}$$

We can substitute for U and express the left side as:

$$\text{InvMixColumns}(\text{InvSubBytes}(T)_i + L_i) \tag{13}$$

Substitute for T and again express the left side as:

$$\begin{aligned} \text{InvMixColumns}(\text{InvSubBytes}(\text{InvShiftRows}(\\ \text{InvMixColumns}(S + K))_i + L_i) \end{aligned} \tag{14}$$

And so on for all n rounds of the cipher, yielding $n - 1$ conditions. We can now use these conditions to find the correct ordering of the bytes in each intermediate state. Once we recover the first state, we can get the plaintext by adding the first round key to it.

In the previous chapter we noted that it may not be possible to get the correct ordering of the whole key, e.g. in AES-256. Instead, we only recovered a set of candidate keys. It's clear we could use them all in the state-ordering calculations above, but that would lead to a significant performance penalty. Instead, we can perform these calculations just for the fourth columns of each state, because we have recovered the most information for these: With AES-128, we know the fourth columns of all round keys except for the last, with AES-256 we know the fourth columns of all round keys except for the first and the last, and with AES-192 we know the fourth columns of each third round's key and we can calculate the others from them. We do not need to know the actual permutation of the key, because we can test for all of them if necessary. By applying these keys to the conditions on states, we can conclusively state which keys could not have led to the observed substitution table accesses.

4 RESULTS AND DISCUSSION

In order to demonstrate this approach we created application `AesSniffer`. It is written in C++ and consists of three main parts: A system-dependent library for performing the debugging and memory access work, a system-independent library for recovering keys and plaintexts and a console tool for performing these tasks on third-party applications. This organization allows for a simple adaptation of the tool to different operating systems: while the supplied application is intended for Microsoft Windows, it is possible to adapt it to other OSes by reimplementing the debugging core and the user interface while keeping the recovery part unchanged – or improve the recovery part and apply it to all variants of the application.

4.1 Library and Application Tests

The tests were performed using Microsoft Windows 7 SP1 x86 in a virtual machine provided by Oracle VM VirtualBox with AES-NI and SSEx instructions disabled. Several popular cryptographic libraries were tested: OpenSSL[2], CryptoPP[3], Botan[4] and WinCrypt[5]; in all cases a simple application for encrypting and decrypting a sample block in the ECB operation mode with a random 128-bit, 192-bit and 256-bit key. As a further test, two existing third-party applications which use their own implementation of AES, were tested: 7-Zip[6] and Putty[7]; both of these libraries use the CTR operation mode. Finally, we tested our application's ability to recover data sent and received by PowerShell's `Invoke-WebRequest` command over the HTTPS protocol using the CBC operation mode.

In all of these cases, the application was successful in recovering both the key and the plaintext, although with some limitations:

4.1.1 OpenSSL

All encryption and decryption of data was successfully detected and all keys and data were recovered. We did encounter 8 unrecognized accesses to the substitution tables due to the cache prefetch code which is a part of the OpenSSL implementation.

4.1.2 CryptoPP

The T-tables used by the library are calculated at runtime from the standard SubBytes tables. While these tables were eventually found by our application, accesses to them detected and data and keys recovered, this process did consume some time during which some encryption was already performed, leading to the loss of the early data. We also noticed 256 unrecognized accesses to the SubBytes table while the T-tables were being constructed.

4.1.3 Botan

Much like CryptoPP, Botan also calculates the T-tables at runtime, leading to the loss of early data before the calculated T-tables could have been found. Other than that, our application was able to detect all encryptions and recover both the keys and the data.

4.1.4 WinCrypt

WinCrypt is a part of the Windows family of operation systems. We were able to recover all keys and data, but we did encounter an error in the library shipped with Windows 7 and Windows 8.1: After the key expansion, four unexpected accesses to the substitution table were encountered, probably as a result of an unnecessary SubWord call for the last column of the expanded key, because the data were successfully recovered regardless. In Windows 10, no such accesses were observed.

[2]https://www.openssl.org/
[3]https://www.cryptopp.com/
[4]https://botan.randombit.net/
[5]https://botan.randombit.net/
[6]https://www.7-zip.org/
[7]https://www.putty.org/

4.1.5 7-Zip

7-Zip supports encryption of the archives using the AES cipher in CTR mode. We performed a test with a file consisting of 32 zero bytes (two AES blocks) in the "Store" mode (without compression). Our application successfully detected two encryptions; both used the same key and the plaintext were two successive counter values (`0x01, 0x00, 0x00, 0x00, ...` and `0x02, 0x00, 0x00, 0x00, ...`), as expected.

4.1.6 Putty

Putty uses the SSH protocol to encrypt the transferred data, and AES is one of the supported ciphers in this protocols. We attempted to recover data from a connection where AES-256 in CTR mode was the agreed-upon cipher between client and server. Two distinct key expansions were detected and recovered as well as a lot of encryptions. After we XORed the recovered plaintexts with the encrypted data captured by WireShark, we were able to observe data structure expected in the unencrypted contents of the SSH protocol.

4.1.7 PowerShell

PowerShell is a scripting language which, among other things, supports reading web data using the HTTPS protocol using the `Invoke-WebRequest` command. When we enforced the use of TLS version 1.0, the client and server agreed upon using the AES-128 cipher in CBC mode. Our application successfully detected both the key expansion and the encryption as well as decryption of data and we were able to verify that the recovered plaintext contained the expected data of the HTTP protocol.

4.2 Performance Tests

During our tests we measured the speed of our application in different scenarios using 7-Zip. The tool was chosen because it allows precise specification of the size of the data as well as precise measurement of time; at the same time it is a real-world application and as such can provide a real-world benchmark, unlike a custom benchmarking application which would be heavily dependent on the actual organization of the AES code (e.g. whether the substitution tables were located in the same memory page(s) as some other frequently used data or code items). We created files of 256, 4096 and 65536 bytes and measured how long did 7-Zip take processing these files without compression but with AES-256 encryption in different scenarios based on our application's settings. The results can be seen in Table 1.

It is apparent that the presence of our application carries a significant performance penalty even if no key- and plaintext-recovery is being done. This is caused by the fact that on any access to the memory page containing a detected substitution table causes a page fault, a number of context switches between the application, our AesSniffer and the operating system, and several `VirtualProtectEx` calls, not to mention the possible need for using a software decoder of the affected instruction. While this penalty can be reduced if the monitored application used a friendlier memory layout (e.g. the substitution tables would be located in dedicated memory pages), the opposite is also possible – if, for example, the substitution table occupies the same memory page as a virtual method table of some frequently used object class, the penalty could be much more pronounced, even more so if some frequently used code was located there as well.

Another significant increase in the processing time can be observed if the detection of AES-192 is activated. The reason for that is that with AES-192 there are far more possible permutations of the key than with AES-128 and AES-256 because the columns of the 192-bit key depend on five other columns rather than three columns with 128- and 256-bit keys.

Finally, the size of the data to be encrypted obviously increases the overall time because more accesses to the substitution table are required – 160 accesses per block in case of AES-128, 192 accesses per block in case of AES-192 and 224 accesses per block in case of AES-256.

While these penalties may seem overwhelming, it should be noted that they are still far more manageable than other dynamic approaches. We implemented a very simple tracer into our application, one which forces single-stepping of all instructions without any additional processing (i.e. no AES detection at all). Encryption of a 256-byte would then take more than 38900 seconds, or something like 200-times the worst case of our code with full detections enabled.

If better performance was desired, it is possible to separate the gathering of data (monitoring accesses to substitution tables) from the processing of the data (key and plaintext recovery): While the first phase must by necessity be performed at the time the accesses are done, the second phase does not need to – it is quite sufficient to process the data asynchronously, e.g. in a different thread or even offline from a record of the accesses in a file. That would at the very least resolve the penalty for using AES-192 which is unnecessarily incurred synchronously in the current implementation.

Table 1: Performance penalty of AES detection in 7-Zip based on the size of encrypted data and the settings for the detection. The "Simple Tracer" scenario represents a minimal trap-after-each-instruction implementation without any additional logic.

File size [B]	256	4096	65536
Time to process [s]			
Without AesSniffer	0.06	0.06	0.15
AesSniffer, no detection	148.83	150.96	166.93
AesSniffer, only AES-256	150.06	182.26	329.56
AesSniffer, AES-256 and 128	159.52	204.53	347.51
AesSniffer, full detection	199.97	662.81	8138.84
Simple tracer	38964.26	–	–

4.3 Limitations

From the presented tests it is obvious that the approach works in general. However, it does have certain limitations from the real-world-usage point of view:

The whole approach is based on a set of assumptions which seem to hold true in may real-world libraries, but that is certainly no guarantee that it would hold for all of them. In particular, with better compilers and more aggressive optimization techniques in them, we may well expect that loop unrolling could violate the "no mixing of rounds" requirement. Similarly, the use of true multithreading for encryption might violate the condition of blocks being processed sequentially, although in this case the use of thread identifiers could be added to the processing code to distinguish table accesses from different threads.

The key- and data recovery process is fairly slow to the point of being unusable in scenarios where a large amount of data is being processed, and it is certainly possible to write code in such a way that the slowdown might become even more pronounced. While the speed could be improved in the general case, against a targeted attack there is little to be done.

The major problem with our approach is that it is only suitable for AES implementations which use substitution tables located in the main memory. Bit-slicing implementations are completely immune to this approach, as is the usage of AES-NI instructions. Fortunately, AES-NI can be readily disabled in current environments, forcing the AES implementations to fall back to the software-based implementation. Also, it might be possible to replace the AES-NI instruction codes in memory with an `INT3` instruction to effect a breakpoint and then read the arguments from the respective registers, although this approach suffers from other issues (i.e. it is difficult if not impossible to avoid false positives and the whole approach is vulnerable to various anti-debugging techniques). The bit-slicing implementations seem completely immune even against this option.

It should be noted that even with a traditional software implementation of AES, our algorithm may run into trouble if the substitution tables do not exist in the actual executable and instead are precalculated during the program's runtime. The less time there is between the precalculation and the use of the tables, the more likely it is that some encryption may escape the detection. The applications which only perform one task and then quit may be particularly prone to this issue. Research is needed to establish what, if anything, can be done about it.

5 CONCLUSION

In this paper we proposed to introduce an algorithm which can automatically detect the use of AES cipher and to automatically recover both the key and the plaintext. Our approach is based on the observation that traditional software implementation of AES make use of precalculated substituted tables which can be detected in the application's memory, and that by evaluating the accesses to these tables we can deduce the desired information. While this approach carries a significant performance penalty and does not work against more hardware-based implementations such as Bit-slicing or the use of AES-NI, it still succeeds in a number of situations: We verified that we are able to recover key and plaintext with several commonly used encryption libraries using our own test applications, and we demonstrated that we could do the same with existing third-party applications, two of which use their own custom implementation of the AES cipher. It can be expected that other applications would be vulnerable to this approach as well, particularly so if they offload the encryption work to the libraries we tested.

At the moment, there is little to be done with the AES implementations which do not use a substitution table. However, further research may provide some ways of overcome this problem. It may well be worth the while to try to distinguish AES-NI instructions

from the rest of the machine code and non-code data in the code segment and introduce breakpoints in their place. That remains to be seen. Similarly, further research may enable us to discover ways of automatically recovering keys and plaintexts of other encryption algorithms, although the widespread use of AES seems to make that a lower priority than the support for AES-NI.

Even though our implementation does not work against these special cases, it seems to work well enough in practice. After all, AES-NI can be purposefully disabled, if we desire so, and when that's been done, our tool can be used to automatically recover keys and plaintext data in many real-world scenarios, giving the users a simple way of observing the encrypted traffic which would otherwise be difficult to replicate. We believe that is a worthwhile contribution.

ACKNOWLEDGEMENTS

The authors acknowledge the support of the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16_019/000 0765 "Research Center for Informatics".

REFERENCES

Auxier, B., Rainie, L., Anderson, M., Perrin, A., Kumar, M., and Turner, E. (2019). Americans and privacy: Concerned, confused and feeling lack of control over their personal information.

Daemen, J. and Rijmen, V. (2002). *The design of Rijndael: AES – the Advanced Encryption Standard*. Springer-Verlag, Berlin, Heidelberg, 1st edition.

DeBlasio, J. (2020). Protecting users from insecure downloads in google chrome.

Gueron, S. (2010). Intel Advanced Encryption Standard (AES) New Instructions Set. Technical report, Intel Corporation.

Intel Corporation (2019). *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*. Intel Corporation.

Käsper, E. and Schwabe, P. (2009). Faster and timing-attack resistant AES-GCM. *Cryptographic Hardware and Embedded Systems – CHES 2009*, pages 1–17.

Malbrain, K. (2007). Higher performance AES C byte-implementation.

Matsui, M. (2006). How Far Can We Go on the x64 Processors? *Fast Software Encryption*, pages 341–358.

Polyakov, A. (2016). crypto/aes/asm/aes-586.pl. [online].

Rijmen, V., Bosselaers, A., and Barreto, P. (2000). Optimised ANSI C code for the Rijndael cipher.