Automatic Detection and Decryption of AES Using Dynamic Analysis

Josef Kokeš1 · Jonatan Matějka1 · Róbert Lórencz1

Received: 29 September 2021 / Accepted: 21 May 2022 / Published online: 17 June 2022
© The Author(s) 2022

Abstract
In this paper we propose a set of algorithms that can automatically detect the use of AES and automatically recover both the encryption key and the plaintext, assuming that we can control the code flow of the encrypting program, e.g., when an application is performing encryption without the user’s permission. The first algorithm 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. We also discuss the options of recovering the same data when hardware-assisted AES implementations on Intel-compatible architectures are used.

Keywords AES · Rijndael · Cipher · Encryption · S-Box · Key recovery · Plaintext recovery · Dynamic analysis · Side channel attack

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 [1], or in the depreciation 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 [2].

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 that makes use of the dynamic analysis to recover the sensitive data generated or used by a third-party application: By exploiting the fact that applications do have a significant level of control over their child processes which essentially creates a particularly powerful side channel, we propose a tool that would run an encrypted application (the
target) as its subprocess and manipulate its control flow in such a way that the AES encryption keys and even the plaintexts are revealed. The process is fully automatic and highly universal, at the cost of performance.

Note: This is an extended version of paper [3]. “Detecting and Recovering AES by Interception the AES-NI Instructions” and “Detecting and Recovering AES by Intercepting the HW-Assisted Bit-Slicing Algorithm” sections represent the majority of this extension, while the other sections may contain some minor updates but are mostly kept intact.

Common Implementations of AES

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

AES is probably the most commonly used block cipher in the world. Its design and structure is defined in Ref. [4] 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 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), one of the following approaches is usually taken:

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 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 using 6 pre-calculated tables with the same structure.

Add-round-key is again usually implemented in a straight-forward 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 Ref. [5].

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 Ref. [4]:

\[
\begin{pmatrix}
    d_{0j} \\
    d_{1j} \\
    d_{2j} \\
    d_{3j}
\end{pmatrix} = \sum_{i=0}^{3} T_i(a_{i,j}+C) + \begin{pmatrix}
    k_{0j} \\
    k_{1j} \\
    k_{2j} \\
    k_{3j}
\end{pmatrix},
\]

(1)

where + is the operation XOR, \(C_i\) are the respective left shifted 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

\[
T_0(x) = \begin{pmatrix}
    2 \cdot S[x] \\
    1 \cdot S[x] \\
    1 \cdot S[x] \\
    3 \cdot S[x]
\end{pmatrix},
T_1(x) = \begin{pmatrix}
    3 \cdot S[x] \\
    2 \cdot S[x] \\
    1 \cdot S[x] \\
    1 \cdot S[x]
\end{pmatrix},
\]

\[
T_2(x) = \begin{pmatrix}
    1 \cdot S[x] \\
    3 \cdot S[x] \\
    2 \cdot S[x] \\
    1 \cdot S[x]
\end{pmatrix},
T_3(x) = \begin{pmatrix}
    1 \cdot S[x] \\
    1 \cdot S[x] \\
    3 \cdot S[x] \\
    2 \cdot S[x]
\end{pmatrix},
\]

(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 [6].

Implementation Using Bit-Slicing

The bit-slicing implementation [7] 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 [8]. 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.

### Implementation Using AES-NI

In 2008, Intel introduced a new instruction set extension for a hardware support of AES encryption and decryption [9]. 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 comparison 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.

### Detecting and Recovering AES Through the Use of S-Box

Let’s first tackle the purely software-based approaches, i.e., the naïve implementation and the T-tables implementation. In both cases the application makes use of pre-calculated tables during the actual encryption and decryption, although the tables themselves may be dynamically calculated using the cipher’s specification during the crypto engine initialization. Our approach is based on attaching to the target application as a debugger and then making use of the debugging APIs to monitor data accesses to these tables with the intention to deduce both the key and the data from the order and the precise location of the accesses.

To achieve this goal, four subproblems need to be resolved: First we need to locate the substitution tables in the process’ memory. Then we need to establish a method for getting notified about the process trying to access these tables. When that is done, we need a technique for determining the type of the access (during encryption/decryption, during key expansion, and irrelevant accesses due to other concerns), and finally we need a method for putting it all together and extracting the actual sensitive data from the accesses. In fact, it turns out that the last two subproblems are closely related and only using them in combination can reveal the full information.

### Locating Tables

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.

### 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 [10] 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

---

\(^1\) 1, 2 and 3 for encryption tables and 1, 9, 11, 13 and 14 for decryption tables.
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.

**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 Ref. [11]. 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 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.

**Monitoring Key Expansion**

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

\[
W_i = \begin{cases} 
K_i & \text{for } i < N_k \\
W_{i-N_k} + r(s(W_{i-1})) + rcon_i/N_k & \text{for } i \geq N_k, i \equiv 0 \pmod{N_k} \\
W_{i-N_k} + s(W_{i-1}) & \text{for } i \geq N_k, i \equiv 4 \pmod{N_k} \\
W_{i-N_k} + W_{i-1} & \text{otherwise}
\end{cases}
\]

(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:

\[
r(w) = r \begin{pmatrix} w_0 \\
  w_1 \\
  w_2 \\
  w_3 \\
  w_4 \\
  w_5 \\
  w_6 \\
  w_7 \\
  w_8 \\
  w_9 \\
  w_{10} \\
  w_{11} \\
  w_{12} \\
  w_{13} \\
  w_{14} \\
  w_{15} \\
\end{pmatrix}
\]

(4)

\[
s(w) = s \begin{pmatrix} w_0 \\
  w_1 \\
  w_2 \\
  w_3 \\
  w_4 \\
  w_5 \\
  w_6 \\
  w_7 \\
  w_8 \\
  w_9 \\
  w_{10} \\
  w_{11} \\
  w_{12} \\
  w_{13} \\
  w_{14} \\
  w_{15} \\
\end{pmatrix}
\]

\[
rcon_i = \begin{pmatrix} 2^{i-1} \\
  0 \\
  0 \\
  0 \\
\end{pmatrix}, \text{ for } i \in \mathbb{N}^+
\]

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 Fig. 1. Then:
Fig. 1 Dependency of columns in the expanded 128-bit key. The grayed boxes describe the final expression [3]

\[ W_i = W_{i-4} + W_{i-1} = \]
\[ = W_{i-8} + W_{i-5} + W_{i-2} = \]
\[ = W_{i-12} + W_{i-9} + W_{i-6} + W_{i-3} = \]
\[ = W_{i-16} + W_{i-13} + W_{i-10} + W_{i-7} + r(s(W_{i-4})) + rcon_{(i-3)/4} = \]
\[ = W_{i-16} + r(s(W_{i-4})) + rcon_{(i-3)/4} \]

(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:

\[ W_{19} = W_3 + r(s(W_{18})) + rcon_4 \]
\[ : \]
\[ W_{39} = W_{23} + r(s(W_{38})) + rcon_9 \]

(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 key expansion process or the assumptions above have been violated.

In this fashion we 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:

\[ W_{41} = W_5 + W_{17} + W_{29} + r(s(W_{35})) + rcon_6 \]
\[ W_{47} = W_{11} + W_{23} + W_{35} + r(s(W_{41})) + rcon_7 \]

(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:

\[ W_{39} = W_7 + s(W_{35}) \]
\[ W_{47} = W_{15} + s(W_{43}) \]
\[ W_{55} = W_{23} + s(W_{51}) \]
\[ W_{43} = W_{11} + r(s(W_{39})) + rcon_5 \]
\[ W_{51} = W_{19} + r(s(W_{47})) + rcon_6 \]

(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 \( 4^4 = 24 \) 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.

**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

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

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

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

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:

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

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. For example, 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

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

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

$$\text{InvMixColumns}(\text{InvSubBytes}(T_i) + L_i)$$

Substitute for $T$ and again express the left side as

$$\text{InvMixColumns}(\text{InvSubBytes}(\text{InvShiftRows}(\text{InvMixColumns}(S + K)_i) + L_i))$$

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.

**Results and Discussion**

To demonstrate this approach we created application AesS-niffer. 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.

**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, CryptoPP, Botan* and WinCrypt; 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

---

* https://botan.randombit.net/.
2 https://www.openssl.org/.
3 https://www.cryptopp.com/.
4 https://botan.randombit.net/.
5 https://botan.randombit.net/.
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.

**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.

**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.

**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.

**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.

**7-Zip:** A popular compression utility 7-Zip supports encryption of the archives using the AES cipher in the 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 \(0x01, 0x00, 0x00, 0x00, \ldots\) and \(0x02, 0x00, 0x00, 0x00, \ldots\), as expected.

**Putty:** A GUI re-implementation of the SSH protocol for Windows, Putty uses AES (and other ciphers) to encrypt the transferred data between the client and the server. We attempted to recover data from a connection, where AES-256 in the 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.

**PowerShell:** Microsoft 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 the 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.

### 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.

### Table 1 Performance penalty of AES detection in 7-Zip based on the size of encrypted data and the settings for the detection

| 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 | – | – |

The “Simple Tracer” scenario represents a minimal trap-after-each-instruction implementation without any additional logic.

---

\(^6\) https://www.7-zip.org/

\(^7\) https://www.putty.org/
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 s, 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.

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. Of course, if an application supports these hardware-assisted modes, it will tend to prioritize them over the S-box based approach. If we want to ensure that our monitoring technique will be successful, we must somehow convince the application that neither vector extensions (usually SSE or SSE2) nor the AES-NI extensions to the basic instruction set are available. That can sometimes be done in the computer setup (the older BIOS or the newer EFI setup), where the support for these features can be disabled, but with the advancing adoption of these features in new software that ability has been steadily disappearing over time and nowadays many systems do not provide it at all. Given the expected hardware requirements of the upcoming Microsoft Windows 11 operating system [12], we can assume that sooner rather than later it will not even be possible to start the operating system if these features are disabled.

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.

Detecting and Recovering AES by Intercepting the AES-NI Instructions

The approach described in the previous section is not suitable to applications which make use of the AES-NI instructions. At the same time, the growing availability of these instructions in new CPUs make them increasingly interesting to applications that want to use AES for encryption. For this reason, it is imperative that a technique for recovering
plaintexts and encryption keys used with these instructions is needed.

**AES-NI Instructions**

The AES-NI consists of six new instructions that were added to the instruction set [9]:

**AESENC xmmA, xmmB/m128**: This instruction performs one full round of the AES encryption. On input, the cipher’s state is stored in one XMM register and the round key is stored either in another XMM register or in a memory location. After execution, the first XMM register will contain the modified state, which becomes the input state for the next round of the cipher.

**AESENCLAST xmmA, xmmB/m128**: This instruction behaves similar to AESENC, except that it skips the MixColumns step. That makes the instruction suitable for use in the last encryption round which does not include that step. The output of the instruction becomes the final ciphertext.

**AESDEC xmmA, xmmB/m128** and **AESDECNCLAST xmmA, xmmB/m128** behave similarly to the respective AESENC and AESENCLAST instructions, except that the decryption steps are performed instead of the encryption steps.

**AESKEYGENASSIST xmmA, xmmB/m128, imm8**: This instruction is used during the key expansion to generate the round keys required by the cipher. It will take 128 bits of the previous round’s encryption keys in the second argument and calculate 128 bits of the next round’s encryption key to store into xmmA. The immediate value represents the first (non-zero) value of the \( rcon \) constant for that round.

**AESIMC xmmA, xmmB/m128**: This instruction performs the inverse MixColumns operation on an encryption key stored in the second argument and saves the result to the first argument. The instruction is intended for the generation of the decrypted key suitable for the AESDEC and AESDECLAST instructions as the instructions make use of the “Equivalent Inverse Cipher” ordering of round operations rather than the “Inverse Cipher” ordering provided by the official specification of the cipher.

**Key and Plaintext Recovery**

The specifications of the instructions allow for a very efficient key and plaintext recovery, provided that the instruction use can be detected.

**Key Recovery**

If we learn of the execution of AESKEYGENASSIST, we learn two round keys; by evaluating the third argument, we learn which round key was being generated, because each round uses a different constant. Thus the key recovery algorithm can be very simple and self-sufficient: Wait for one (for a 128-bit key) or two (192-bit and 256-bit key) subsequent AESKEYGENASSIST operations, saving both their input values. Use the \( rcon \) value to determine the round key number and perform the respective number of inverse key generation steps to learn the master key. Optionally, the values of one additional instruction use can be saved to detect whether the same key was used for all calls. That would also allow us to determine the size of the key, by verifying how large a part of the round key can be calculated from its first 128 bits.

**Plaintext Recovery**

By capturing the execution of AESDECLAST instruction we directly learn the plaintext (the output of the instructions), regardless of the key size.

Recovering the plaintext during encryption is somewhat more complex and depends on the size of the key.

For a 128-bit key, it is sufficient to capture the AESENCLAST call and save both the final ciphertext and the round key used to generate it. By reversing the key generation process, we can then calculate the round keys for all previous rounds and perform decryption of the ciphertext. We need to know that 128-bit keys are used, though; that information can come from the key generation process or from some external application-specific source, e.g., the metadata of the TLS protocol.

For 192-bit or 256-bit keys, where their size is not known, a different approach is needed: We need to capture at least two successive AESENC calls and verify that they are indeed successive, that is, the output of the first call becomes the input of the second call. That gives us 256 bits of round keys which we can once again use to recover the full AES key and determine its size and then use it to decrypt the plaintext.

However, in this scenario we also need to determine the round number of the round we captured. We can learn the information by monitoring the AESKEYGENASSIST instruction and matching the actually used keys to the values calculated from the key schedule. Another approach is to monitor all AESENC calls and assume that any call, where the value of xmmA does not match the value of xmmA of the previous call is the beginning of the first round after the initial AddRoundKey operation. Yet another approach is to monitor for the sequence of AESENC and AESENCLAST, where the output of the first instruction becomes the input of the second one, because then we know we have the data of the last two rounds; that does not necessarily provide us with the specific round number, but in the worst case scenario we

---

* A register used by the vector unit of the Intel architecture.
can offer all three possibilities (9th and 10th, 11th and 12th, or 13th and 14th round) to the user to select the correct one. Note that if an application uses AES-NI, it almost guarantees that the developer had to manually write code in assembler. If that was the case, it is to be expected that the developer performed multiple optimizations. For example, in the OpenSSL implementation of AES in CCM mode, the AES rounds for the encryption of data are intertwined with the AES rounds for calculating the MAC value, thus violating the assumption from the previous section about non-overlapping encryption operations. The recovery process needs to take that possibility into consideration.

**Monitoring the Use of AES-NI**

All of the recovery techniques discussed above, however, are predicated by our ability to detect that an AES-NI instruction was executed. Unfortunately, that is rather difficult, because these instructions are not privileged in any way (to be detected by a non-permitted privilege instruction use) and can read all their data from registers (preventing any kind of memory breakpoint, even if we could determine in advance which memory they would use, which is unlikely in any case).

In the paper [13], a very nice approach to this problem was suggested: The authors’ technique involves the use of a CPU without the AES-NI instruction set while simultaneously tricking the application by faking the return value of the CPUID instruction (they achieve that using a virtual machine hypervisor to capture the CPUID call, because this instruction is privileged) to believe that AES-NI is in fact available. That convinces the application to use the AES-NI instructions which are unknown to the CPU and thus the CPU itself generates a hardware exception when these instructions are executed. The control application can capture this exception, decode the offending instruction, save all of its inputs and then emulate it in software, letting the target application continue as if nothing happened.

This approach is very simple, powerful and efficient, because it involves no modification of the target applications code (which could otherwise be detected) and the instruction detection is done by the CPU itself as a by-product of its standard functionality. As a result, the application runs full-speed most of the time and only when an AES-NI instruction is executed some processing is necessary. However, as explained above, this processing is quite simple as far as our needs are concerned, so we could adopt this approach to our tool almost without changes. In fact, we can even avoid faking the CPUID result value, because we have a solution for the software-only AES encryption and decryption. However, there is a significant performance penalty, and to overcome that we need a CPU that does not support AES-NI, which is becoming increasingly difficult these days, or a motherboard which can disable the functionality, which has the same problem.

In the absence of a CPU lacking AES-NI, there remain two main approaches to the problem:

Traditional emulation could be used to create such a CPU. Unlike virtualization, where the target application for the most part runs on the actual hardware (with some modifications done by the virtualization layer) and makes use of its features, emulation has a full control over the behavior of the CPU used by the target application. Thus it is definitely possible, especially in case of open-source emulators, to selectively remove some instructions from the emulated instruction set. The disadvantage of this approach is, of course, speed—emulation is extremely slow compared to native and even the virtualized execution.

In scenarios, where performance is important, conventional execution breakpoints can be used. The monitoring tool would need to search the virtual memory space of the target application for the machine code of the specific instructions and then use either a hardware or a software breakpoint to stop execution when such an instruction is encountered. On the Intel Architecture, a software breakpoint is more viable in this usage, because we will need to break on far too many instructions than the hardware breakpoint limits allow—e.g., a typical AES encryption will involve 9, 11 or 13 AESENC calls followed by an AESDECLAST call. Unfortunately, this approach carries all the drawbacks of software breakpoints, including an easy detection and the risk of overwriting an instruction which is a part of another instruction rather than an AES-NI instruction—although the latter is somewhat mitigated by the fact that the machine code of the AES-NI instructions is quite long and rather distinctive as shown in Table 2.

### Table 2: Machine code of the AES-NI instructions [10]

| Instruction       | Machine code |
|-------------------|--------------|
| AESENC            | 0x66 0x0F 0x38 0xDC r/m |
| AESENCLAST        | 0x66 0x0F 0x38 0xDD r/m |
| AESDEC            | 0x66 0x0F 0x38 0xDE r/m |
| AESDECLAST        | 0x66 0x0F 0x38 0xDF r/m |
| AESIMC            | 0x66 0x0F 0x38 0xDB r/m |
| AESKEYGENASSIST   | 0x66 0x0F 0x3A 0xDF r/m imm8 |

**AES-NI and S-Box Combinations**

It has been suggested to us that the AESKEYGENASSIST instruction can be slower than software implementations and that software developers may well opt to implement AES in

---

9 For performance reasons the rounds are almost always loop-enrolled if the AES-NI instructions are being used.
such a way as to use AES-NI for the conventional encryption/decryption and to use another approach (e.g., the S-box software implementation) for key expansion. While we are not aware of any such software and as a result cannot test how our algorithm would perform in such circumstances, we are confident that it would still work. This confidence is based on the fact that our approach was designed and tested to strictly separate the monitoring process from the process of recovery of sensitive data.

It is true that both our key-expansion recovery algorithm and our data recovery algorithm require that each respective operation is not “mixed” with another operation (as stated in “Monitoring Key Expansion” and “Monitoring Encryption” sections) and encryption requires that the key generated in the last key-expansion algorithm is used. However, if the specified conditions were met and the necessary data were captured, the actual implementation of the operations is not important—e.g., the encryption monitoring algorithm only needs the captured key, regardless of its source.

Taking this fact into consideration, monitoring a combination of AES implementations might even prove to be easier than monitoring a dedicated one, because the requirements that “no other access to substitution tables is performed” while executing either key-expansion or encryption would be automatically satisfied if only one of the operations (usually the key-expansion) were actually using the S-box.

However, this is only an expectation at this moment. It needs to be thoroughly tested first before being considered a fact.

**Detecting and Recovering AES by Intercepting the HW-Assisted Bit-Slicing Algorithm**

Compared to the support for AES-NI instructions, the need to automatically detect bit-slicing is much reduced. It is supported by the encryption libraries (e.g., OpenSSL) and as such it can be used by applications to perform the encryption and decryption, but generally if AES-NI is available, the libraries tend to default to it, because the performance is much better. Unless the user takes specific steps to prevent the use of AES-NI altogether (e.g., to alleviate concerns about the security of the instructions themselves), bit-slicing will only get used on CPUs which do support vector instructions but do not support AES-NI; since all Skylake-based or newer Intel CPUs (introduced in 2015) and Jaguar-based or newer AMD CPUs (introduced in 2013) support AES-NI, and since AES-NI has been available in all but the least powerful CPUs, since at least 2011, this is a dwindling concern.

However, we can encounter bit-slicing implementations in the wild and we need to consider that they will remain in use even in the future. Could we detect their use automatically and could we extract the sensitive data (plaintexts, keys) used in them?

The answer to that question is unclear at the moment. It is definitely the case that bit-slicing is much more difficult to detect than any other implementation covered in this paper, because there is no universal detectable component in them: The traditional pure-software implementations depend on the availability of the substitution tables, which need to reside in the memory if the implementation should not be prohibitively slow. The AES-NI implementations make use of the AES-NI instructions which have a rather distinctive machine code. However, bit-slicing implementations do not depend on either: they do not use substitution tables (which are instead represented by logical operations) and while the instructions used are specific in that vector instructions (MMX or SSE) are used, these instructions are also used for other purposes so they can not be summarily considered an indication of the AES use. It is only when the instructions are performed in a specific order that they comprise AES, but detecting such a scenario seems very difficult, if not impossible.

One viable option seems to be to focus specifically on popular implementations. These are relatively rare, probably due to the fact that a bit-slicing implementation requires manually writing a rather complex assembler code which is a daunting task for many developers. Thus it can be expected that most applications that use the technique will in fact contain the same or at least very similar code which may be detectable, e.g., through the use of signatures. For example, the OpenSSL implementation of bit-slicing AES builds around the code created by Mike Hamburg, and this code is possibly detectable through the use of constants, as seen in the sources [14]. Other implementations may be detectable through the use of signatures, much like antivirus tools detect malware. Code sequences discovered in this way could possibly be replaced with a call to a function which were injected into the target application’s code by the control application, which would save the inputs and then simulate or even copy the original code. However, this approach seems to be highly specific and unreliable and we find it difficult to consider it on par with the much more generic nature of the approaches proposed previously.

It may be that a more generic scheme is available, but it’s an open question whether it is even worth the effort to search for it, considering the usage aspects mentioned above. We should keep in mind that unlike a traditional attacker who gets his execution environment thrust upon herself, we are operating from a position of having a complete control over the environment. Thus, if we did encounter code which would prefer bit-slicing to AES-NI, we could attempt to disable SSE instructions as well as AES-NI, thus forcing the use of a purely software-based implementation of AES.
Conclusions

In this paper we proposed to introduce algorithms which can automatically detect the use of AES cipher and to automatically recover both the key and the plaintext. Our approach was primarily 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.

With the software-based implementations thus resolved, we switched our focus to the hardware-assisted AES implementations which are not vulnerable to our approach, because they do not use a substitution table. We discussed options of achieving the same goals with them, albeit with different means, and found that AES-NI use may well be vulnerable to the use of traditional breakpoints. Certainly, once an AES-NI instruction has been identified and the code stopped at the beginning of the instruction, the process of extracting the key or the plaintext becomes relatively simple, especially compared to the rather complicated calculations of interactions between the key and the cipher state in our previous approach. Given the rather distinctive form of the AES-NI instructions’ machine code, this approach seems viable.

Bit-slicing approaches, on the other hand, present a much more difficult challenge. At the moment we are unable to propose a truly universal approach to their detection, much less to the universal recovery of sensitive data entering the algorithm. An approach exploiting the specific implementations was tentatively suggested, but whether it is viable in achieving the same results as the previous approaches remain to be seen.

However, it should be noted that even if we are forced to limit the technique to the case of a pure-software implementation, the results are quite impressive: By the increasingly difficult but still somewhat possible expedient of disabling the AES-NI extensions of the CPU, we were able to automatically recover both the key and the plaintext from several commonly used encryption libraries as well as several applications sporting their own custom implementation of AES. That suggests that our tool can be used to do the same thing in many real-world scenarios and provide the users with a simple way of observing the encrypted traffic which would otherwise be difficult to replicate. We believe that is a worthwhile contribution.

Author Contributions Chapters 2 and 3 of the article are based on a master’s thesis by JM under the supervision of JK, revised and adapted for the publication in article form by JK. Remaining chapters as well as the paper preparation, proofing and presentation were done by JK. All work was consulted and supervised by RL.

Funding The authors acknowledge the support of the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16_019/0000765 “Research Center for Informatics”.

Availability of Data and Materials Not applicable.

Code Availability Code available upon request.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

Ethics approval Not applicable.

Consent to participate Not applicable.

Consent for publication Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. DeBlasio J. Protecting users from insecure downloads in google chrome. https://blog.chromium.org/2020/02/protection-users-from-insecure.html. Accessed 28 Sept 2020.

2. Auxier B, Rainie L, Anderson M, Perrin A, Kumar M, Turner E. Americans and Privacy: concerned, confused and feeling lack of control over their personal information. https://www.pewresearch.org/internet/2019/11/15/americans-and-privacy-concerned-confused-and-feeling-lack-of-control-over-their-personal-information/. Accessed 28 Sept 2020.

3. Kokeš J, Matějka J, Lórencz R. Automatic detection and decryption of AES by monitoring S-Box access. In: Proceedings of the 7th international conference on information systems security and
privacy-ICISSP, pp. 172–180. SciTePress, Setúbal (2021). https://doi.org/10.5220/0010255201720180.

4. Daemen J, Rijmen V. The design of Rijndael: AES—the advanced encryption standard. 1st ed. Berlin, Heidelberg: Springer; 2002.

5. Malbrain K. Higher performance AES C byte-implementation. https://www.geocities.ws/malbrain/aesstable2_c.html. Accessed 28 Sept 2020.

6. Rijmen V, Bosselaers A, Barreto P. Optimised ANSI C code for the Rijndael Cipher. https://github.com.openssl/openssl/blob/master/crypto/aes/aes_x86core.c. Accessed 28 Sept 2020.

7. Käser E, Schwabe P. Faster and timing-attack resistant AES-GCM. Cryptographic hardware and embedded systems—CHES 2009. 2009. p. 1–17. Accessed 28 Sept 2020.

8. Matsui M. How far can we go on the x64 processors? Fast software encryption. 2006. p. 341–358 Accessed 28 Sept 2020.

9. Gueron S. Intel Advanced Encryption Standard (AES) New Instructions Set. Technical report, Intel Corporation. 2010. https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf. Accessed 28 Sept 2020.

10. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B: System Programming Guide, Part 2. Intel Corporation. 2019. Intel Corporation. https://software.intel.com/sites/default/files/managed/7c/1f/253669-sdm-vol-3b.pdf.

11. Polyakov A. crypto/aes/asm/aes-586.pl. [online]. 2016. https://github.com/openssl/openssl/blob/master/crypto/aes/asm/aes-586.pl. Accessed 28 Sept 2020.

12. Microsoft Corporation: Windows Processor Requirements. https://docs.microsoft.com/en-us/windows-hardware/design/minimum/windows-processor-requirements. Accessed 1 Aug 2021.

13. Takehisa T, Nogawa H, Morii M. Aes flow interception: Key snooping method on virtual machine—exception handling attack for aes-ni-. IACR Cryptology ePrint Archive. 2011;2011:428.

14. Hamburg M. Constant-time SSSE3 AES core implementation. https://github.com/openssl/openssl/blob/master/crypto/aes/asm/vpaes-x86.pl. Accessed 1 Aug 2021.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.