A Systematic Study of Cache Side Channels across AES Implementations

Heiko Mantel[1], Alexandra Weber[1], and Boris Köpf [2]

[1] Computer Science Department, TU Darmstadt, Darmstadt, Germany
mantel@cs.tu-darmstadt.de, weber@mais.informatik.tu-darmstadt.de

[2] IMDEA Software Institute, Madrid, Spain
boris.koepf@imdea.org

Abstract While the AES algorithm is regarded as secure, many implementations of AES are prone to cache side-channel attacks. The lookup tables traditionally used in AES implementations for storing precomputed results provide speedup for encryption and decryption. How such lookup tables are used is known to affect the vulnerability to side channels, but the concrete effects in actual AES implementations are not yet sufficiently well understood. In this article, we analyze and compare multiple off-the-shelf AES implementations wrt. their vulnerability to cache side-channel attacks. By applying quantitative program analysis techniques in a systematic fashion, we shed light on the influence of implementation techniques for AES on cache-side-channel leakage bounds.

1 Introduction

The Advanced Encryption Standard (AES) is a widely used symmetric cipher that is approved by the U.S. National Security Agency for security-critical applications [8]. While traditional attacks against AES are considered infeasible as of today, software implementations of AES are known to be highly susceptible to cache side-channel attacks [5, 15, 18, 19, 32]. While such side channels can be avoided by bitsliced implementations [6, 23], lookup-table-based implementations, which aim at better performance, are often vulnerable and wide spread.

To understand the vulnerability to cache side-channel attacks, recall that the 128bit version of AES relies on 10 rounds of transformations. The first nine rounds consist of the steps *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundkey*. The last transformation round is similar but skips the step *MixColumns*. Many cryptographic libraries, such as LibTomCrypt [37], mbed TLS [3], Nettle [28] and OpenSSL [30], precompute the results of applying *SubBytes*, *ShiftRows*, and *MixColumns* for all possible inputs. They store the precomputed results in four 1KB lookup tables with entries of 32bit each. With this, the AES rounds can be implemented by simple table lookups to indices depending on the current state – which is beneficial for performance but introduces the cache side channel.

While the table organization of the first nine rounds specified by [10] is used as the default in most implementations, there is a variety of approaches for implementing the last round of AES:

– mbed TLS and Nettle rely on an additional 0.25KB table with 8bit entries to store the S-Box for *SubBytes*.
– OpenSSL computes the results of *SubBytes* and *ShiftRows* based on the lookup tables for the main rounds.
– LibTomCrypt uses four additional 1KB lookup tables with 32bit entries for the last round.

The organization of the lookup tables affects the vulnerability of AES implementations to cache side-channel attacks. While this effect was observed early on [32, 34] and studied based on an analytical model [39], it has not yet been analyzed based on the actual target of the attack, which is the executable code.

In this article, we use program analysis techniques for a systematic, quantitative study of cache-side-channel leakage across AES implementations. We analyze executable code of AES implementations from LibTomCrypt, mbed TLS, Nettle, and OpenSSL. More concretely, we systematically derive upper bounds for the leakage of these executables to a number of adversaries that are commonly considered in the literature. We also describe the effects of table preloading and of varying the cache configuration (i.e. cache size and replacement strategy) across implementations. By our study, it becomes clear how the usage of lookup tables in AES implementations influences the height of leakage bounds within the same cache size and across cache sizes. For instance, the leakage bounds for the lookup-table-based implementations stabilize with increasing cache size. The stabilization occurs once the cache is large enough for the mapping from AES memory blocks (dominated by the lookup tables) to cache sets to be injective.

We used the CacheAudit static analyzer [12] as a starting point for our study. Analyzing the AES executables in their original form required us to extend the tool's scope in terms of, both supported x86 instructions and CPU flags. This required significant engineering effort. The extended CacheAudit is available under `www.mais.informatik.tu-darmstadt.de/cacheaudit-essos17.html`.

2 Preliminaries

In this section, we review the necessary background on AES, caches, and cache side-channel attacks.

AES. AES is a widely used symmetric block cipher proposed by Daemen and Rijmen (originally as "Rijndael") [10]. The AES algorithm operates in multiple transformation rounds. It depends on the AES key's size, how many rounds are performed. The inputs to each round are the current state of the transformed message or ciphertext and a *round key* that is generated from the AES key by a key expansion. Each round consists of multiple steps, including a substitution step that can be implemented using a lookup table of size 0.25KB, called *S-Box*. Lookup tables of size 1KB are often used to store precomputed results of entire transformation rounds for all possible inputs to speed up the computation. The result for a given input is then retrieved using an index to the lookup table [10].

Lookup-table-based implementations are available in many libraries, including LibTomCrypt, mbed TLS, Nettle, and OpenSSL. However, AES can also be

implemented without using lookup tables. For instance, OpenSSL defaults to AES-NI, i.e., AES encryption with hardware support using dedicated x86 instructions, and the AES implementation in NaCl is based on bitslicing, which implements the AES transformation rounds on the fly.

In our study, we consider the lookup-table-based AES implementations from LibTomCrypt, mbed TLS, Nettle, and OpenSSL for a key size of 128bit, which implies that ten transformation rounds are performed [10]. The AES implementations in mbed TLS and Nettle AES use an S-Box and four 1KB lookup tables. OpenSSL AES also uses four 1KB lookup tables but no S-Box, while LibTomCrypt AES uses eight 1KB lookup tables and no S-Box.

Caches. Caches are small and fast memories that store copies of selected memory entries to reduce the average memory access time of the Central Processing Unit (CPU). If the CPU accesses a memory entry that resides in the cache, it encounters a *cache hit*, and the entry is quickly read from the cache. Otherwise, the CPU encounters a *cache miss*, and the entry needs to be fetched first from the main memory to the cache, which is significantly slower.

In our study, we consider 4-way set-associative caches with 64Byte line size and FIFO replacement. The chunks in which memory entries can be loaded into the cache are called *memory blocks*. Memory blocks are cached in *cache lines* of the same size as memory blocks, namely the *line size*. The *associativity* of a cache defines how many cache lines form one *cache set*. A given memory block is mapped to one specific cache set but can be cached in any of the cache lines in this cache set. A cache with associativity k is called k-way set-associative. If a memory block shall be cached into a cache set that is full, then another memory block is evicted from the cache set according to a replacement strategy, e.g., to evict the least recently cached memory block (*FIFO*).

Cache side-channel attacks. A side-channel attack recovers information about inputs to a program from characteristics of program runs. In 2002, Page [33] showed that the interaction between a program and the cache is such a characteristics, i.e., one that can be used to mount a side-channel attack. Such attacks are known as *cache side-channel attacks*. Since table lookups in lookup-table-based AES implementations depend on the secret key, they are prone to different kinds of attacks: *Time-based attacks* [5] recover the secret key from measurements of the the overall execution time; *access-based attacks* [15,32] recover the secret key from the cache state after termination; and *trace-based attacks* [1] recover the key from sequences of cache hits and misses.

3 Our Approach

We analyze AES implementations wrt. potential cache side channel,s based on information theory and static analysis. The static-analysis tool that we employ is an extension of CacheAudit [12] that we developed for this research project. Our extensions increase the tool's coverage of the x86 machine language and improve the tool's precision wrt. its treatment of processor flags. These changes were

crucial for analyzing the four AES implementations, without having to modify their off-the-shelf source code, and they might be beneficial for others.

We describe our approach to side-channel analysis in Section 3.1, illustrate it using mbed TLS AES as an example in Section 3.2, and sketch our conceptual and technical extensions of CacheAudit in Section 3.3.

3.1 Static Bounds on Cache Side Channels

A common approach for quantifying the information leaks of a program is to compute (upper bounds on) the number of observations that an adversary can make. This number comes with different interpretations in terms of security, such as lower bounds for the expected number of guesses required for recovering a secret [26] or upper bounds on the probability for correctly guessing the secret in one shot [38]. Moreover, this number can be obtained by combining off-the-shelf static analysis with model counting techniques [4, 29]. The computation of this number has been implemented based on abstract interpretation [24], bounded model checking [16], and symbolic execution [36].

The CacheAudit static analyzer [12] leverages this basic idea for quantifying cache side channels of x86 executables, based on abstract interpretation. Given an x86 executable, CacheAudit computes bounds wrt. the three adversary models discussed in Section 2. Namely,

- for *access-based* attackers (denoted acc), CacheAudit computes a set O^{acc} that contains all possible states of the cache after termination. Here, cache states are represented as tuples of sequences of memory blocks, where each sequence is of bounded length and represents the content of one cache set.
- for *trace-based* attackers (denoted tr), CacheAudit computes a set $O^{\mathsf{tr}} \subseteq \{hit, miss\}^*$ that contains all possible traces of cache hits and misses that can occur in an execution.
- for *time-based* attackers (denoted time), CacheAudit computes a set $O^{\mathsf{time}} \subseteq \mathbb{N}$ that contains the possible execution times that can occur.

In addition, CacheAudit computes a set O^{accd} that contains a representation of cache states as tuples of integers, where each integer represents the amount of blocks loaded in a particular cache set. This corresponds to the possible observations of a fourth attacker, the *blurred access-based* attacker (denoted accd). This attacker is similar to acc, in the sense that it shares the cache with the victim, but it does not share the memory with the victim.

Bounds on the the information (in bits) leaked to the four adversaries are given by $\log_2 |O^a|$ for $a \in \{\mathsf{acc}, \mathsf{tr}, \mathsf{time}, \mathsf{accd}\}$.

3.2 Analysis of AES from mbed TLS

We illustrate our approach using the AES implementation from mbed TLS (previously known as PolarSSL). This library is used, e.g., in the implementation of OpenVPN [31] and of Internet-of-Things products [2].

Cache [KB]	2	4	8	16	32	64	128
Leakage [bit] $\leq$	92.6	114.5	91.8	71.2	69.6	69.6	69.0
Entropy [bit] $\geq$	163.4	141.5	164.2	184.8	186.4	186.4	187

Table 1: Bounds for mbed TLS on leakage and entropy under acc

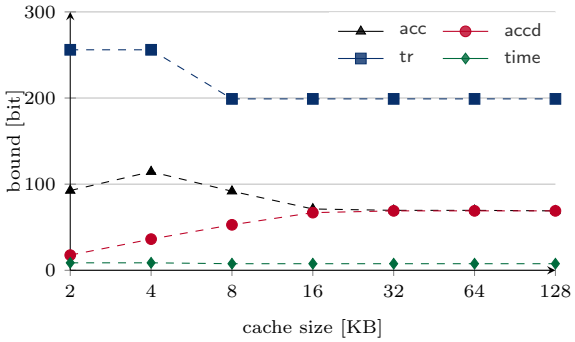

Figure 1: Leakage bounds for mbed TLS AES encryption

Computation of bounds for mbed TLS *under acc.* The leakage bounds computed by our extension of CacheAudit for mbed TLS under acc for different cache sizes are listed in Table 1.[3] Table 1 also contains lower bounds on the remaining min-entropy of the 256 secret bits (key and message) after a side-channel observation. In the remainder of the article, we only explicate leakage bounds because the entropy can be easily computed. It equals 256bit minus the leakage bounds.

Note that the acc leakage bounds converge to 69bit, starting between the cache sizes 16KB and 32KB. This could be due to the fact that at least 17KB of cache are required for an injective mapping from memory blocks (dominated by 4.25KB lookup tables) to cache sets. Once the cache is so big that each cache set can contain at most one memory block, an attacker under acc is able to infer exactly which blocks are accessed by mbed TLS AES. In practice, a convergence of leakage bounds with growing cache sizes implies that leakage bounds will remain valid when processors with larger caches become available in the future.

Exploration of other attacker models. The leakage bounds that we obtain for mbed TLS AES, using all four attacker models, are depicted in Figure 1.[4]

We observe that the leakage bounds for all attacker models converge with increasing cache size. This suggests that the leakage bounds for the mbed TLS AES implementation are robust against future hardware with larger caches, not only with respect to acc, but also for the attacker models accd, tr, and time.

[3] We round all leakage bounds up to one decimal place and truncate them to the maximum leakage of 256bit (128bit message and 128bit key) throughout the article.

[4] To support the reader in reading such diagrams, we connect the leakage bounds computed for adjacent cache sizes and the same attacker model by dashed lines.

The leakage bounds for acc and accd converge to the same value, namely 69bit. This could be due to the fact that an attacker under accd (like under acc) can infer exactly which memory blocks are cached – given that the mapping from memory blocks to cache sets is injective. In practice, this means that, for a system running mbed TLS AES, reducing the attack surface from acc to accd does not lead to better leakage bounds – given that the system has a cache of more than 16KB. In contrast, much better leakage bounds can be achieved when reducing the attack surface to time (e.g., the 7.7bit computed for a cache size of 128KB correspond to 3% of the key and message only).

Before convergence, the leakage bounds computed for tr and time (marked by -■- and -♦-, respectively) decrease with increasing cache size. We will get back to this point in Section 4. In contrast, the leakage bounds for accd (marked by -●-) increase with increasing cache size. Interestingly, the evolution of leakage bounds for acc (marked by -▲-) follows a different pattern. These bounds increase, until a peak is reached for 4KB cache size, and then decrease, until they stabilize. The peak observable in the acc bounds could be due to the exact fit of memory blocks with mbed TLS AES data (dominated by 4.25KB lookup tables) into a cache with size around 4.25KB. If the memory blocks fit exactly into the cache, the most information can be conveyed through the ordering of memory blocks, so that the potential leakage to an attacker under acc is maximized.

Note that our quantitative analysis of mbed TLS AES not only allows us to observe the influence of implementation-level design decisions on the interplay between cache size, attack surface, and leakage bounds. It also enables us to speculate about the practical consequences in an informed manner and, hence, to shed more light on the effects of such design choices. In Section 4, we study the influence of implementation-level design choices on the interplay between cache size, attack surface, and leakage bounds for three further off-the-shelf AES implementations. Moreover, we compare the effects of implementation-level design choices across all four AES implementations.

Details on our analysis setup. In our analysis of mbed TLS AES, we considered the sequential composition of the key expansion function `mbedtls_aes_setkey_enc` with the encryption function `mbedtls_aes_encrypt` from the file `aes.c` of mbed TLS version 2.2.1, configured to use hard-coded tables (option MBEDTLS_AES_ROM_TABLES) and to not use hardware support (option MBEDTLS_PADLOCK_C). We compiled to a 32bit x86 binary without additional code for overflow protection (option −fno−stack−protector).

We configured the AES implementation to use a key size of 128bit. Note that this choice complies with the recommendation of the US National Institute of Standards and Technology. They recommend a security strength of at least 128bit to protect sensitive data in unclassified applications beyond the year 2031 [13, Section 5.6.2]. For simplicity, we configured the message size in the AES implementation also to 128bit. We configured CacheAudit to assume a four-way set-associative cache with 64Byte line size. This cache configuration is used, e.g., for the level 2 cache in the current Intel micro-architecture Skylake [17, Table 2-4]. We used FIFO replacement and varied the cache size from 2KB to 128KB.

Remark 1. A previous version (1.3.7) of mbed TLS AES was already analyzed in [11] (with focus on key size 128bit) and in [12] (with focus on key size 256bit). Like our analysis, [12] considered encryption jointly with key generation, while [11] considered encryption only. In [11] and [12], mbed TLS AES was transformed before the analysis to meet the x86 sublanguage supported by the analysis tool. As usual, a code transformation was chosen that preserves the code's semantics.

We extended the analysis tool to support the analysis of mbed TLS AES without code transformations. The reader might wonder how our results for the off-the-shelf binaries for Version 2.2.1 compare to the ones in [11, 12] for the transformed code snippets of mbed TLS (Version 1.3.7). In brief, our results are rather similar (including the convergence of the bounds for acc and accd from a cache size of 16KB). This similarity shows that both the evolution of mbed TLS versions and the application of code transformations in [11], [12] did not have substantial effects (neither positive nor negative) on the leakage bounds.

3.3 Tool Support

From the beginning of our study, we wanted to analyze off-the-shelf AES implementations in their original form (i.e., without code transformations like the one discussed at the end of Section 3.2). To make this possible, support for additional x86 instructions was needed in CacheAudit. We have added such support. We extended the x86 parser and the abstract x86 semantics in CacheAudit for the instructions listed in Table 2. Now, CacheAudit supports all instructions that occur in the relevant code snippets from the off-the-shelf binaries of LibTomCrypt, mbed TLS, Nettle, and OpenSSL AES.

Some of the binaries contain jump instructions that branch on the sign flag or the overflow flag, which both were previously not supported by CacheAudit. For instance, the conditional jump instruction Jnle (opcode 0F8F) occurs in the binary of mbed TLS AES encryption, and the conditional jump instruction Jl (opcode 0F8C) occurs in the binary of LibTomCrypt AES decryption. Jnle branches on the previously supported zero flag and the previously unsupported sign flag. Jl branches on both previously unsupported flags, i.e., the sign flag and

Type	New Instructions
Arithmetic	2D (Sub), 18 (Sbb), 19 (Sbb), 11 (Adc), F7/6 (Div), 3C (Cmp)
Logic	08 (Or), 30 (Xor), 84 (Test), A9 (Test), F6/0 (Test)
Bitstring	0FA4 (Shld), 0FA5 (Shld), 0FAC (Shrd), 0FAD (Shrd)
Stack	07 (Pop)
Jump	7C, 0F8C, 7D, 0F8D, 70, 0F80, 71, 0F81, 78, 0F88, 79, 0F89, 7E, 0F8E, 7F, 0F8F (all Jcc)
Move	0F48 (Cmovs)

Table 2: Extended language coverage in CacheAudit

the overflow flag. In abstract interpretation, both branches of a conditional need to be considered if the abstraction is too imprecise to determine which branch must be chosen. This can lead to substantial imprecision of analysis results. To avoid such imprecision, we conceptually revised the abstraction employed by CacheAudit and modified the implementation of CacheAudit to support this abstraction. The new abstraction represents the states of the sign and overflow flag on the abstract level with high precision. To implement this abstraction, we refined the data structure for representing flags on the abstract level and adapted the implementation of the abstract semantics of all x86 instructions.

The resulting, extended CacheAudit version enabled us to analyze the off-the-shelf binaries for mbed TLS AES, with the results described in Section 3.2. The extended version of CacheAudit is also the basis for our systematic study of cache side channels across AES implementations reported in Sections 4 and 5.

Remark 2. In our study, we focus on architectures with a single cache. We leave a thorough analysis of multiple cache levels to future work. In particular, the effects of different cache inclusion policies deserve to be studied in detail.

While we focus on the FIFO cache replacement strategy throughout this article, we also investigated other replacement strategies, namely LRU (least recently used) and PLRU (Pseudo-LRU). We observed that the replacement strategies influence the concrete leakage bounds. The cache sizes at which the leakage bounds stabilize also vary across the replacement strategies. Interestingly, the leakage bounds for 128KB cache size are identical for all three replacement strategies. We leave a more extensive investigation of replacement strategies for future work.

4 Leakage across AES Implementations

The technique of lookup tables that store precomputed round transformations is supported by popular libraries like OpenSSL and mbed TLS. We investigate four such implementations, namely LibTomCrypt, mbed TLS, Nettle and OpenSSL AES. All four implementations use four 1KB lookup tables with 32bit entries to store the precomputed transformations for the first nine AES rounds. The implementation of the last AES round, which uses a different transformation, differs across the implementations. OpenSSL AES reuses the existing four lookup tables for the last AES round, while mbed TLS and Nettle AES use an additional 0.25KB S-Box with 8bit entries, and LibTomCrypt AES uses four additional 1KB lookup tables with 32bit entries. In this section, we study the effects of this design choice on the cache-side-channel leakage. We investigate how the different implementations of the last round compare in terms of

- security guarantees against cache side channels and
- the interplay between cache sizes and security guarantees.

To this end, we compute leakage bounds on the AES implementations from LibTomCrypt, mbed TLS, Nettle, and OpenSSL (see Table 3 for the exact functions that we analyze), with the experimental setup described in Section 3.2.

Our results suggest that using fewer additional lookup tables in the last round of AES leads to better security guarantees against attackers under acc

library	configuration	analyzed functions
LibTomCrypt 1.17	`ENCRYPT_ONLY`, `LTC_NO_ASM`,`ARGTYPE`	`rijndael_enc_setup`, `rijndael_enc_ecb_encrypt` (`aes.c`)
mbed TLS 2.2.1	`MBEDTLS_AES_ROM_TABLES`, removed `MBEDTLS_PADLOCK_C`	`mbedtls_aes_setkey_enc`, `mbedtls_aes_encrypt` (`aes.c`)
Nettle 3.2	default	`aes128_set_encrypt_key` (`aes128-set-encrypt-key.c`), `aes128_encrypt` (`aes-encrypt.c`)
OpenSSL 1.0.1t	default	`private_AES_set_encrypt_key`, `AES_encrypt` (`aes_core.c`)

Table 3: AES implementations for which leakage bounds were computed

and accd. Furthermore, our security guarantees for implementations with fewer additional tables are more robust against an increase of the cache size. They stabilize already at a smaller cache size. In the subsequent subsections, we discuss the influence of the lookup tables in the last AES round in detail.

4.1 Security Guarantees

To study the influence of the lookup tables in the last AES round on the height of leakage bounds, we focus on a fixed cache size of 128KB.

The lookup tables in AES implementations have been considered with respect to access-based attackers by Osvik, Shamir, and Tromer [32]. They discuss the use of smaller lookup tables (e.g., one 1KB lookup table or one 2KB lookup table in the main rounds of AES) as a countermeasure to access-based attacks. They state that for certain access-based attackers "smaller tables necessitate more measurements by the attacker", i.e., reduce the leakage of one program run. The leakage bounds that we obtain for the access-based attacker models (listed in Table 4) confirm this. For both accd and acc, we obtain the lowest leakage bounds, namely 64bit, for OpenSSL AES, which uses only 4KB of lookup tables. The implementations from mbed TLS and Nettle AES, which use lookup tables with a total size of 4.25KB, follow closely with leakage bounds of 69bit. The leakage bounds for LibTomCrypt AES, which uses lookup tables with twice the total size, namely 8KB, are roughly twice as high, namely 129bit. Interestingly, reducing the total size of lookup tables in one transformation round only, already has a positive effect on the leakage bounds. LibTomCrypt AES and mbed TLS AES use lookup tables of the same total size in the first nine rounds, but differ in the total size of lookup tables used in the last round. While mbed TLS AES uses a single additional S-Box of 0.25KB in the last round, LibTomCrypt AES, which has significantly higher leakage bounds, uses four additional lookup tables that each require 1KB.

	LibTomCrypt	mbed TLS	Nettle	OpenSSL
accd	129bit	69bit	69bit	64bit
acc	129bit	69bit	69bit	64bit

Table 4: acc/accd leakage bounds for 128KB cache

	LibTomCrypt	mbed TLS	Nettle	OpenSSL
time	7.7bit	7.7bit	7.7bit	7.7bit
tr	198bit	199bit	199bit	196bit

Table 5: time/trace leakage bounds for 128KB cache

The influence of lookup tables in AES implementations on time- and trace-based attackers has been studied by Page [34]. He recommends the use of S-Boxes with 8bit entries, instead of lookup tables with 32bit entries. Page argues that, the smaller table entries are, the more table entries share the same cache line. Consequently, for smaller table entries, cache hits and misses reveal less information. Tiri, Aciiçmez, Neve, and Andersen [39] confirm this for two time-based attacks in a practical evaluation of variants of OpenSSL AES. They compare an attack on OpenSSL AES, which reuses the 1KB lookup tables with 32bit entries in the last round, to an attack on a variant of OpenSSL AES that uses an S-Box with 8bit entries in the last round. The latter attack requires more attacker measurements than the former. Our leakage bounds for the attacker models time and tr are listed in Table 5. We observe that the leakage bounds are very similar across the different implementations. In particular, the bounds for mbed TLS and Nettle, which use S-Boxes with 8bit entries in the last round, are not lower than the bounds for LibTomCrypt and OpenSSL, which use tables with 32bit entries in the last round. Note that, in our approach, we approximate the possible observations about cache hits, but not the value that an individual observed cache hit has for the attacker. This difference between our approach and the one in [39] might be the reason for the difference in the findings.

In summary, our study suggests that the use of fewer additional lookup tables in the last round of AES leads to better leakage guarantees against attackers under acc and accd. While a more fine-grained approach would be needed to study the effectiveness of smaller table entries as a countermeasure against trace- and time-based attackers, the leakage bounds are precise enough to confirm that smaller lookup tables are effective against access-based attackers.

4.2 Interplay of Cache Size and Security Guarantees

The leakage bounds that we obtain for varying cache sizes for the attacker models accd and acc are depicted in Figure 2c and Figure 2d.[5] For all four AES implementations the leakage bounds stabilize with increasing cache size. The cache

[5] For LibTomCrypt AES and 2KB cache size, the analysis ran out of memory.

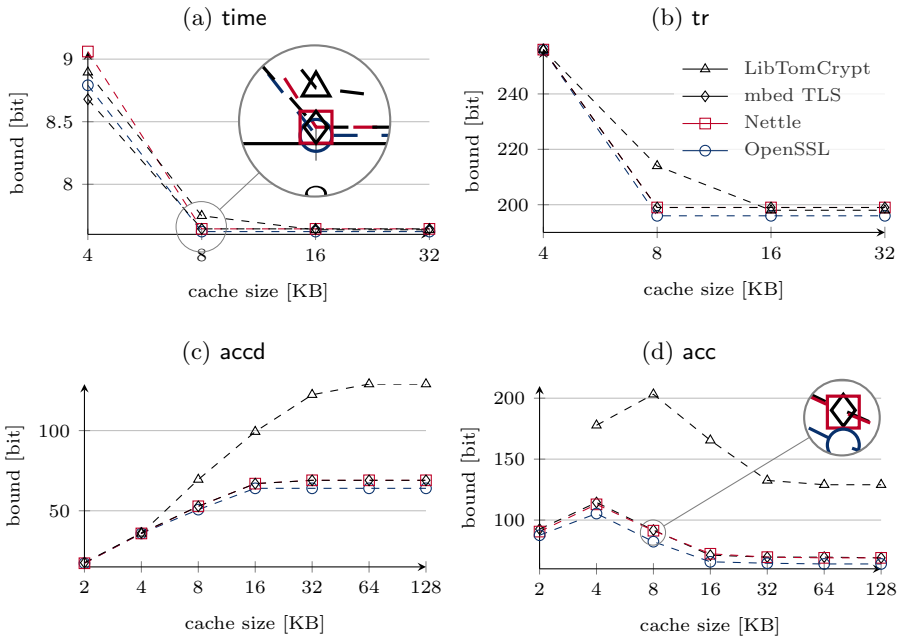

Figure 2: Leakage bounds wrt. the four attacker models

size from which they stabilize differs across the implementations. This could be due to the minimum amount of cache sets that is needed for an injective mapping from memory blocks to cache sets. For LibTomCrypt, which uses 4KB of additional tables in the last AES round, additional 15KB of 4-way set-associative cache are needed, compared to mbed TLS AES, which uses only 0.25KB of additional tables in the last AES round. Note that, it is of practical relevance that the leakage bounds for acc and accd stabilize at the observed points. If leakage bounds are computed for a stabilization point, they are robust against increasing cache sizes, and cache sizes tend to grow with technological improvements. In our analysis, the stabilization point is reached, once the mapping from memory blocks to cache sets is injective.

The leakage bounds that we obtain for the attacker models time and tr are depicted in Figure 2a and Figure 2b. We observe that the tr leakage bounds for all four AES implementations decrease and then stabilize to roughly 200bit with increasing cache size. The bounds for mbed TLS, Nettle, and OpenSSL are stable starting between cache size 4KB and 8KB. The bounds for LibTomCrypt are stable starting between cache size 8KB and 16KB. Note that, for all four implementations, the cache sizes at which the bounds stabilize correspond to the amount of data used by the implementations (dominated by the lookup tables). Since the AES implementations perform 200 accesses to lookup tables during one key expansion and encryption, a leakage of 200bit corresponds to a leakage

of 1bit (hit or miss) per access to a lookup table. The additional leakage before stabilization could be due to secret-dependent eviction of other local variables. Once all memory blocks fit into the cache, local variables are not evicted any more. In practice, a smaller total size of additional lookup tables in the last round of AES leads to more robustness of our tr and time leakage bounds against changes in the cache size.

Overall, our study suggests that the decision how many additional lookup tables are used in the last round of AES has an influence on the robustness of the security guarantees for all four attacker models with respect to future hardware. A smaller total size of additional tables leads to implementations that are robust starting from a smaller cache size.

5 Hardening across AES Implementations

Hardening techniques aim to reduce the side-channel leakage of implementations. The preloading hardening technique is tailored specifically to lookup-table-based implementations of AES. It preloads all memory blocks that belong to lookup tables into the cache, before running the actual implementation. Cache locking [27] locks memory blocks in cache lines. Locked memory blocks cannot be evicted from the cache. If lookup tables are preloaded and then locked in the cache, their presence in the cache is independent of secret information[6].

Does the implementation of the last AES round in a lookup-table-based implementation influence the effectiveness of preloading as a hardening technique? To address this question, we compute leakage bounds for preloading in multiple lookup-table-based AES implementations. These implementations differ in the techniques used to implement the last round of AES. More concretely, we analyze the implementations from LibTomCrypt (last round with four 1KB lookup tables), mbed TLS and Nettle (last round with one 0.25KB S-Box), and OpenSSL (last round without additional lookup tables), to which we manually added preloading. Throughout this section, we assume that no other processes affect the cache content.[7]

In Table 6, each line corresponds to one AES implementation with preloading. The symbol ✓ marks the cache sizes for which we obtain the leakage bound 0bit. The table is the same under all four attacker models acc, accd, tr, and time.

We observe that the leakage bounds for LibTomCrypt stabilize to 0bit for caches greater than 8KB and the leakage bounds for the other AES implementations stabilize to 0bit for caches greater than 4KB. This could be due to the cache size required to hold all lookup table entries and additional variables of an AES implementation. When no lookup-table entry can be evicted from the cache, the cache trace and the final cache state are constant for any secret key and message. To rule out evictions, OpenSSL AES requires at least 4KB cache

[6] Without cache locking, the preloaded table entries might be evicted from the cache by other processes [21, 22].

[7] This can be realized using static cache locking if the cache size exceeds the total size of tables. One could consider dynamic cache locking [27] if the cache is too small.

Cache Size [KB]	4	8	16	32	64	128
LibTomCrypt			✓	✓	✓	✓
Nettle		✓	✓	✓	✓	✓
OpenSSL		✓	✓	✓	✓	✓
mbed TLS		✓	✓	✓	✓	✓

Table 6: Preloading effectiveness for acc/accd/tr/time

for its 4KB lookup tables. The AES implementations from mbed TLS and Nettle require roughly 0.25KB additional cache for the additional S-Box that they use in the last AES round. LibTomCrypt AES requires at least 4KB of additional cache for its additional 4KB of lookup tables in the last round. In practice, this suggests that the use of fewer additional lookup tables in the last round of AES not only makes preloading more efficient (because fewer blocks need to be preloaded), but also makes preloading effective on more systems.

Furthermore, for each AES implementation and cache size, we either obtained the leakage bound 0bit for all attacker models or for none of the attacker models. This could be because the final cache state and the cache trace can depend on secret information if and only if preloaded table entries might be evicted from the cache. In practice this suggests that, if preloading is used, no additional effort has to be spent to reduce the attack surface of an AES implementation from tr to a more restricted attacker model.

Overall, our study suggests that preloading is effective against acc, accd, tr and time for lookup-table-based AES implementations whose data fits into the cache entirely.

Remark 3. It is also possible to avoid cache-side-channel leakage by implementing AES without lookup tables. Instead of precomputing the round transformations, they can be computed on the fly, e.g., using bitslicing. We analyze the bitsliced AES implementation from the library NaCl[8] [6]. We obtain the leakage bound 0bit for all four attacker models and all six cache sizes.

6 Related Work

Cache attacks on AES. AES implementations have been attacked using different techniques to exploit cache-side-channel vulnerabilities.

Bernstein's time-based attack on OpenSSL AES [5] exploited that a given byte of the AES key can be characterized by the running times it induces on different messages. Information about an unknown key was obtained by comparing the duration of multiple sample AES runs with this key against previously measured running times for known keys. The attacker model time captures the

[8] the sequential composition of the functions `crypto_stream_beforenm` (`beforenm.c`) and `crypto_stream_xor_afternm` (`xor_afternm.c`) from NaCl in version 20110221

observations from one sample AES run, where the actual running time is approximated based on the numbers of cache hits and cache misses. Acıiçmez and Koç presented trace-based attacks on OpenSSL AES [1]. The underlying samples of cache traces were generated by instrumenting OpenSSL AES to store all access indices. The attacker model tr captures the observations from one such sample.

Osvik, Shamir, and Tromer mounted access-based cache attacks on OpenSSL AES using two techniques [32]. In Evict+Time, the attacker clears a cache set after running AES and times a subsequent encryption. In Prime+Probe, the attacker fills the cache with his data and times his own accesses to his data in a cache set after an encryption. Both techniques allow an attacker to determine whether a given cache set was used. This scenario is generalized by the attacker model accd, under which attackers can observe the fill-degree of all cache sets.

An asynchronous access-based attack on AES was mounted by Gullasch, Bangerter, and Krenn [15] with a technique later extended to Flush+Reload [42]. These attacks motivated the attacker model acc, which is weaker because it captures a synchronous attacker, who can only observe the final cache state.

While cache side-channel attacks [1, 5, 15, 18, 19, 32, 40] have often targeted OpenSSL, recently the Java library Bouncy Castle was also attacked through a cache side channel [25]. A detailed survey of microarchitectural side-channel attacks is provided by Ge, Yarom, Cock, and Heiser [14].

Hardening techniques for AES. Preloading is a code-based technique to counter cache side channels in lookup-table-based implementations. The cache-locking technique, mentioned in Section 6, is supported by multiple commercial processors [27]. Multiple other code-based, hardware-based, and operating-system-based countermeasures exist. A survey of countermeasures is provided in [14].

Already in 2003, Page considered a variety of code-based countermeasures against trace- and time-based cache side channels, including preloading and lookup tables with smaller entries [34]. As countermeasures against access-based cache side channels, Osvik, Shamir, and Tromer suggested different possibilities to avoid memory accesses [32]. As alternatives to avoiding memory accesses, they discussed, e.g., the use of smaller lookup tables. Brickell et al. suggested to harden AES implementations against cache side channels by permuting the lookup tables during the algorithm and by using one compact lookup table that can be preloaded before each AES round [7]. Crane et al. proposed a randomization of the control flow and the execution characteristics of binaries [9].

On the operating system level, Page considered restricting access to precise timing information, randomizing the duration of memory accesses, and out-of-order execution of memory accesses [34]. On the hypervisor level, StealthMem [20] counters cache side-channels by avoiding that different VMs evict each other's cache lines. Hardware-based countermeasures include larger cache lines, physical shielding of devices [34], and special cache architectures [35, 41].

Leakage across implementations. To our knowledge, ours is the first systematic study of cache-side-channel leakage across off-the-shelf AES implementations.

Different variants of one specific AES implementation have been investigated by Tiri, Acıiçmez, Neve, and Andersen [39]. They propose an analytical model

for time-based cache attacks that predicts the number of required running time samples for key recovery. They validate the model with respect to three variants of OpenSSL AES (5KB, 4.25KB, and 4KB lookup tables), two specific approximations of the attacker model time, and two different cache line sizes. To this end, they mount attacks with an attacker who can directly access the number of cache misses. In their predictions as well as in their attacks, the 4.25KB variant requires more samples than the 5KB variant. While Tiri, Acıiçmez, Neve, and Andersen consider only OpenSSL AES, we investigate cache-side-channel leakage across multiple AES implementations. Furthermore, while Tiri et al. focus on time-based attacks, our study also covers access- and trace-based attacks.

Variants of mbed TLS AES 1.3.7 with/without preloading and varying key sizes (128bit, 192bit, 256bit) have been analyzed by Doychev, Köpf, Mauborgne, and Reinecke [12]. They observed that, under FIFO replacement, preloading is effective against acc, accd, tr, and time for the cache sizes large enough to hold all AES lookup tables. They also observed a positive effect of larger cache sizes on their leakage bounds for accd, tr, and time as well as a negative effect on their bounds for acc. They describe that the acc leakage bounds converge to the same value as the accd leakage bounds because each cache set can contain at most one lookup table block at the point of convergence. Our study shows that the observations from [12] carry over to a newer version of mbed TLS. Moreover, we show that these observations are also valid for three further AES implementations.

7 Conclusion

We conducted a systematic study of cache side channels in off-the-shelf AES implementations, namely OpenSSL, LibTomCrypt, mbed TLS, and Nettle AES. Our goal was to better understand the influence of implementation details on upper bounds for the cache-side-channel leakage of AES implementations.

Our findings suggest that the total size of lookup tables in an AES implementation plays an important role for the leakage bounds on cache side channels. The use of a dedicated S-Box in the last round of AES, for instance, can be avoided by masking entries of the lookup tables used in the first rounds of AES.

An interesting direction for future work will be to study the influence of multiple cache levels and of cache inclusion policies. We hope that the approach used for AES in this article will also be adopted by others to enable the analytic study of cache side channels in a broad range of cryptographic implementations.

Acknowledgements We thank Clémentine Maurice and the anonymous reviewers for helpful comments. We also thank Artem Starostin for inspiring discussions in the initial phase of this project and Xucheng Yin for his contributions to CacheAudit. This work has been funded by the DFG as part of the project Secure Refinement of Cryptographic Algorithms (E3) within the CRC 1119 CROSSING and was supported by Ramón y Cajal grant RYC-2014-16766, Spanish projects TIN2012-39391-C04-01 StrongSoft and TIN2015-70713-R DEDETIS, and Madrid regional project S2013/ICE-2731 N-GREENS.

References

1. Aciiçmez, O., Koç, Ç.K.: Trace-Driven Cache Attacks on AES (Short Paper). In: ICICS. pp. 112–121 (2006)
2. ARM Ltd.: ARM buys Leading IoT Security Company Offspark as it Expands its mbed Platform. https://www.arm.com/about/newsroom/arm-buys-leading-iot-security-company-offspark-as-it-expands-its-mbed-platform.php (2015), [Online; accessed Feb-11-2017]
3. ARM Ltd.: mbed TLS (Version 2.2.1-gpl). https://tls.mbed.org/download/mbedtls-2.2.1-gpl.tgz (2016), [Online; accessed Jul-28-2016]
4. Backes, M., Köpf, B., Rybalchenko, A.: Automatic Discovery and Quantification of Information Leaks. In: S& P. pp. 141–153 (2009)
5. Bernstein, D.J.: Cache-timing attacks on AES. Tech. rep., University of Illinois at Chicago (2005)
6. Bernstein, D.J., Lange, T., Schwabe, P.: The Security Impact of a New Cryptographic Library. In: LATINCRYPT. pp. 159–176 (2012)
7. Brickell, E., Graunke, G., Neve, M., Seifert, J.: Software mitigations to hedge AES against cache-based software side channel vulnerabilities. IACR Cryptology ePrint Archive pp. 1–17 (2006)
8. Committee on National Security Systems: CNSS Policy No. 15: National Information Assurance Policy on the Use of Public Standards for the Secure Sharing of Information Among National Security Systems. https://www.cnss.gov/CNSS/openDoc.cfm?1858/J1y8IPFvRRvn+ZZBw== (2016), [Online; accessed Dec-29-2016]
9. Crane, S., Homescu, A., Brunthaler, S., Larsen, P., Franz, M.: Thwarting cache side-channel attacks through dynamic software diversity. In: NDSS (2015)
10. Daemen, J., Rijmen, V.: AES submission document on Rijndael, Version 2. http://csrc.nist.gov/archive/aes/rijndael/Rijndael.pdf (1999)
11. Doychev, G., Feld, D., Köpf, B., Mauborgne, L., Reineke, J.: CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In: USENIX Security. pp. 431–446 (2013)
12. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: A tool for the static analysis of cache side channels. ACM Transactions on Information and System Security pp. 4:1–4:32 (2015)
13. Elaine Barker: Nist special publication 800-57 part 1, revision 4: Recommendation for key management - part 1: General. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf (2016)
14. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Journal of Cryptographic Engineering pp. 1–27 (2016)
15. Gullasch, D., Bangerter, E., Krenn, S.: Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In: S&P. pp. 490–505 (2011)
16. Heusser, J., Malacaria, P.: Quantifying information leaks in software. In: ACSAC. pp. 261–269 (2010)
17. Intel Corporation: Intel® 64 and IA-32 Architectures Optimization Reference Manual. Order Number: 248966-032 (2016)
18. Irazoqui, G., Eisenbarth, T., Sunar, B.: S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. In: S&P. pp. 591–604 (2015)
19. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a Minute! A fast, Cross-VM Attack on AES. In: RAID. pp. 299–319 (2014)

20. Kim, T., Peinado, M., Mainar-Ruiz, G.: STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In: USENIX Security. pp. 189–204 (2012)
21. Kong, J., Aciiçmez, O., Seifert, J.P., Zhou, H.: Deconstructing new cache designs for thwarting software cache-based side channel attacks. In: CSAW. pp. 25–34 (2008)
22. Kong, J., Aciiçmez, O., Seifert, J.P., Zhou, H.: Hardware-software integrated approaches to defend against software cache-based side channel attacks. In: HPCA. pp. 393–404 (2009)
23. Käsper, E., Schwabe, P.: Faster and Timing-Attack Resistant AES-GCM. In: CHES. pp. 1–17 (2009)
24. Köpf, B., Rybalchenko, A.: Approximation and Randomization for Quantitative Information-Flow Analysis. In: CSF. pp. 3–14 (2010)
25. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security. pp. 549–564 (2016)
26. Massey, J.L.: Guessing and entropy. In: ISIT. p. 204 (1994)
27. Mittal, S.: A Survey of Techniques for Cache Locking. ACM Transactions on Design Automation of Electronic Systems pp. 49:1–49:24 (2016)
28. Möller, N.: Nettle (Version 3.2). `https://ftp.gnu.org/gnu/nettle/nettle-3.2.tar.gz` (2016), [Online; accessed Jul-28-2016]
29. Newsome, J., McCamant, S., Song, D.: Measuring channel capacity to distinguish undue influence. In: PLAS. pp. 73–85 (2009)
30. OpenSSL Software Foundation: OpenSSL (Version 1.0.1t). `https://www.openssl.org/source/openssl-1.0.1t.tar.gz` (2016), [Online; accessed Jul-28-2016]
31. OpenVPN Technologies, I.: HOWTO. `https://openvpn.net/index.php/open-source/documentation/howto.html` (2017), [Online; accessed Feb-16-2017]
32. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: The Case of AES. In: CT-RSA. pp. 1–20 (2006)
33. Page, D.: Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. IACR Cryptology ePrint Archive pp. 1–23 (2002)
34. Page, D.: Defending Against Cache-Based Side-Channel Attacks. Information Security Technical Report pp. 30–44 (2003)
35. Page, D.: Partitioned cache architecture as a side-channel defence mechanism. IACR Cryptology ePrint Archive pp. 1–14 (2005)
36. Pasareanu, C.S., Phan, Q., Malacaria, P.: Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In: CSF. pp. 387–400 (2016)
37. libtom projects: LibTomCrypt (Version 1.17). `https://github.com/libtom/libtomcrypt/archive/1.17.tar.gz` (2010), [Online; accessed Jul-28-2016]
38. Smith, G.: On the Foundations of Quantitative Information Flow. In: FOSSACS. pp. 288–302 (2009)
39. Tiri, K., Aciiçmez, O., Neve, M., Andersen, F.: An analytical model for time-driven cache attacks. In: FSE. pp. 399–413 (2007)
40. Tromer, E., Osvik, D.A., Shamir, A.: Efficient Cache Attacks on AES, and Countermeasures. Journal of Cryptology pp. 37–71 (2010)
41. Wang, Z., Lee, R.B.: A novel cache architecture with enhanced performance and security. In: MICRO. pp. 83–93 (2008)
42. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. pp. 719–732 (2014)