A Comparative Study of Cache Side Channels across AES Implementations and Modes of Operation

Heiko Mantel, Tim Weißmantel, Marc Fischlin, and Alexandra Weber

Department of Computer Science, TU Darmstadt, Germany
name.lastname@tu-darmstadt.de

Abstract. AES applied in different modes of operation is widely considered secure and used in practice. However, it is well known that many software implementations of AES are vulnerable to cache side channels. In this article we investigate the effect of different modes of operation on the potential cache-side-channel leakage of lookup-table-based AES-128 software implementations. To this end, we apply quantitative static side-channel analysis to obtain upper bounds on the leakage of AES-128 in CBC mode, CTR mode and CCM from *OpenSSL* 1.1.1d, *mbedTLS* 2.16.5 and *Nettle* 3.5. By shedding light on how the security wrt. each mode relates to the security of the other modes and the underlying AES implementation, we provide the first systematic study of the cache side channel leakage of AES across modes of operation.

Keywords: Cache Side Channels · AES · Modes of Operation

1 Introduction

Modern secure online communication heavily relies on symmetric-key encryption. For instance, block ciphers ensure the confidentiality of fixed length messages between parties that share a secret key. Currently, the most prominent block cipher is the Advanced Encryption Standard (AES), which was standardized by the US National Institute of Standards and Technology (NIST) [8] and recommended for use in secure communication protocols such as TLS [42].

So-called side-channel attacks leverage observable side effects of executing a vulnerable program to (partially) recover some confidential information. Examples of possible attack vectors include the execution time [5], the power consumption [7], or the cache usage [17] of the targeted program. Cache-based side channels are particularly dangerous. They exploit the latency differences of accesses to caches shared between the adversary and the victim program. Thus, they can be exploited from software without physical access to the device.

Notably, many software implementations of AES are susceptible to such cache-side-channel attacks [16, 18, 24, 26, 30]. Vulnerable implementations of AES follow a lookup-table-based approach. That is, each application of AES is reduced to a series of key- and message-dependent table accesses. The key is leaked since the access location to the lookup tables induce different caching behavior. Such implementations of AES are offered by the popular crypto libraries *OpenSSL* [47], *mbedTLS* [49] and *Nettle* [51].

While the cache-side-channel leakage of AES for a single encryption has been studied in prior work [35], the topic of successive applications of block ciphers has not been considered yet. How block ciphers are applied repeatedly is governed by the mode of operation. Examples for relevant modes are the two popular confidentiality-only modes Cipher Block Chaining (CBC) and Counter (CTR), as well as the authenticated-encryption-with-additional-data (AEAD) mode Counter with CBC-MAC Mode (CCM). All three modes have been recommended by NIST for their use with AES-128 in TLS 1.2 and 1.3 [42].

Static program analysis is one approach for investigating the potential cache-side-channel leakage of programs. In this article we build on an approach that combines abstract interpretation with information theory to provide upper bounds on the potential side channel leakage [29]. We consider timing-based adversaries that observe the execution time, trace-based adversaries that observe a trace of cache hits and misses, and synchronous access-based adversaries that observe aspects of the cache layout after the execution.

In this article we study the potential side channel leakage of lookup-table based AES when applied in CBC, CTR and CCM. Our contributions are:

– We present a base-line assessment of lookup-table based AES-128 implementations from *OpenSSL* 1.1.1d, *mbedTLS* 2.16.5 and *Nettle* 3.5 and find that the *OpenSSL* variant has lower leakage bounds because it doesn't use an additional lookup-table in the last AES round.
– We improve the understanding of the potential cache-side-channel leakage of *OpenSSL* AES-128 in CBC mode, CTR mode and CCM. For instance, we find that the leakage bounds for CBC and CTR mode are similar, while CCM has higher leakage bounds for trace and timing adversaries. The reason is that CCM applies twice as many AES encryptions.
– We provide a comparison of the potential side channel leakage of AES-128 across CBC, CTR, and CCM implementations from *OpenSSL*, *mbedTLS* and *Nettle*. We show that the differences in potential leakage caused by the implementation of the last AES round are neither masked nor significantly amplified by their application in a mode of operation.

Our vision is that the results presented in this article are used as guidance in the decision process when deciding between different modes of operation. For instance, our results suggest that a user can either choose CBC or CTR mode to minimize their risk towards cache-side-channel attacks or to choose CCM which entails a higher risk while also ensuring the authenticity of the encryption.

Outline In Section 2 and Section 3 we introduce preliminaries. In Section 4 we extended the static analysis. In Section 5 we present base-line results for AES-128, which we expand upon in Section 6 where we present and interpret leakage bounds for AES-128 in CBC mode, CTR mode, CCM. We provide an overview on related work in Section 7 and conclude the article in Section 8.

2 Lookup-table based AES & modes of operation

In this section we introduce AES, two different lookup-table-based optimizations of AES, as well as the three modes of operation CBC, CTR and CCM.

AES is a symmetric-key cryptographic algorithm. That is, both the encryption and decryption algorithm operate on a shared secret key. AES is a block cipher that works on 128 bit plaintext blocks and 128 bit ciphertext blocks respectively. There are three variants of AES with varying secret key sizes: 128 bit, 196 bit, and 256 bit keys.

A mode of operation is an algorithm that describes how a block cipher such as AES is applied repeatedly to encrypt multiple message blocks. Building on modes of operation, authenticated encryption with additional data (AEAD) schemes provide confidentiality via encryption and authenticity by computing a Message Authentication Code (MAC) of the ciphertext and some Additional Authenticated Data (AAD).

Prior to AES the Data Encrypt Standard (DES) [2] was the state-of-the-art block cipher. Both CTR and CBC mode have been originally developed for DES in 1979 [3] and 1980 [4], respectively. As a replacement for DES, AES was proposed by Joan Daemen and Vincent Rijmen in 1999 [6]. AES and its use in CBC and CTR mode was recommended by NIST in 2001 [8, 9]. With AES in mind, CCM was developed and standardized in 2003 [13]. NIST recommended AES in CCM[15] one year later.

2.1 Implementations of AES

Both, the encryption and the decryption algorithm of AES, consist of the same four component functions. They operate on a 4×4 Byte state matrix and are applied in multiple rounds. For each round there is one round-specific key, derived by a so-called key expansion algorithm. Processing a 128 bit key takes 10 rounds, processing a 192 bit key takes 12 rounds, and processing a 256 bit key takes 14 rounds. The component functions are:

– AddRoundKey: addition of the state matrix to the round key
– SubBytes: a non-linear substitution on each Byte of the state matrix
– ShiftRows: a cyclically shift of each row in the state matrix
– MixColumns: a matrix multiplication with a predefined constant matrix.

One encryption starts with one application of AddRoundKey to the plaintext. Then in each round SubBytes, ShiftRows, MixColumns and AddRoundKey are applied successively. The last round differs from all previous rounds by skipping the application of MixColumns.

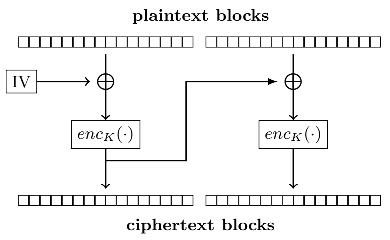
There is a lookup-table-based optimization for processors with a word length of 32 bit. In fact, the SubBytes, ShiftRows and MixColumns component functions can be combined into 16 accesses to 4 pre-computed lookup tables with 256 4 B entries each. This optimization was already published as part of the initial proposal of AES [6].

This optimization, however, is not applicable to the last AES round, because the MixColumns component function is skipped. Implementation details of the last round vary across the three crypto libraries covered in this article. *OpenSSL* follows the initial AES proposal [6] and reuses the existing tables by masking out the effects of MixColumns. The implementations from *mbedTLS* and *Nettle* calculate the last round explicitly by applying an extra 256 B lookup table (called S-Box) for the SubBytes function followed by the circular row shift of ShiftRows.

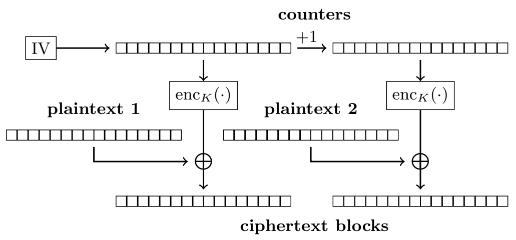
2.2 Structure of CBC mode, CTR mode, and CCM

The general structure of the CBC encryption algorithm is presented in Figure 1a. Each plaintext block is xored with the previous ciphertext block of the previous block. Then the intermediate result is encrypted with secret key K (denoted by $enc_K(\cdot)$) to produce the cipher text. For the first block a randomly chosen initialization vector (IV) is used instead of the previous ciphertext.

In CTR mode, an increasing encrypted counter value is used for a one-time-pad encryption of each plaintext block (shown in Figure 1b). Similar to CBC, a randomly chosen IV is used as the initial value of the counter. For all subsequent blocks, the counter is incremented by one.



(a) CBC mode



(b) CTR mode

Fig. 1: Encryption of two plaintext blocks in CBC and CTR mode.

The encryption algorithm of CCM takes the secret key, a plain text, some AAD and a nonce (number used once) and computes a cipher text and a MAC. In the depiction of an encryption of two message blocks with CCM in Figure 2, we omit the AAD. The length of the nonce and the length of the MAC (denoted with *tlen*) are configurable.

CCM is a combination of CTR mode for encryption with CBC-MAC for authentication following the mac-then-encrypt paradigm. The encryption in CCM differs from CTR mode in the construction of the 128 bit counter value. In concrete terms, in CCM the i-th counter consists of a set of flags and the nonce prepended to the bit representation of the number i. Then the encrypted counter values is used for a one-time-pad encryption of the MAC and the plaintext.

The aforementioned MAC is computed by applying CBC-MAC to the AAD and the plaintext. As depicted in Figure 2, CBC-MAC is essentially an encryption with underlying block cipher in CBC mode where all blocks except for the

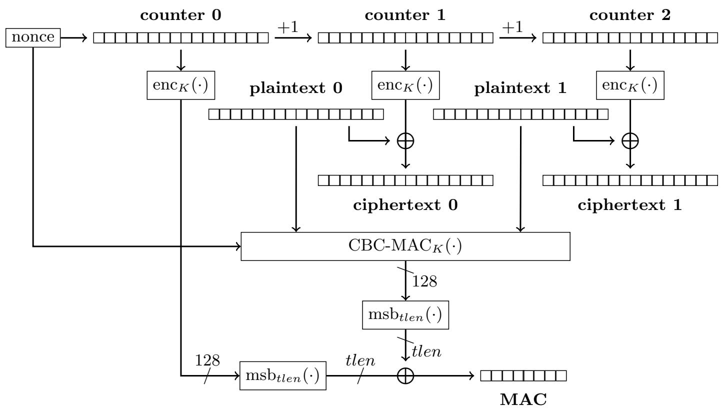


Fig. 2: Authenticated encryption of two plaintext blocks in CCM.

last block are discarded. Since by construction the value of the last block depends on all input blocks, the last block acts as a hash of the input.

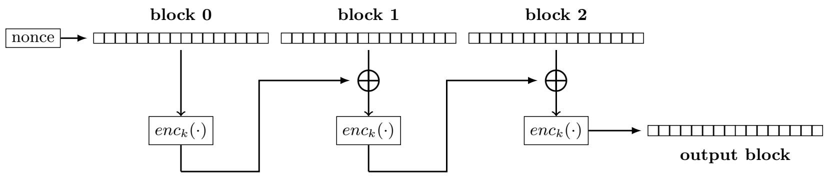


Fig. 3: MAC of three blocks with CBC-MAC.

3 Quantitative analysis of cache-side-channel attacks

In this section, we introduce cache side channels and our approach to static analysis for the quantification of cache-side-channel leakage.

3.1 Caches and cache-side-channel attacks

Modern CPUs make use of fast memory caches, to bridge the substantial discrepancy between the speed of the processor and the speed of main memory. Accesses to data that is already cached (i.e. cache hits) are significantly faster than accesses that have to retrieve the data from memory (i.e. cache misses).

Caches are split into multiple cache sets, which in turn are further split into multiple cache lines. The index of the cache set used to cache a given memory block is derived from the starting address of the block. For instance a 32 KiB

8-way associative (meaning 8 lines per set) cache with 64 B cache lines, has 64 cache sets and uses bits 7 to 12 of the address to determine the cache set.

When new data is loaded into the cache, one cache line from the targeted cache set is evicted if all cache lines are already occupied. Which cache line is chosen for eviction is determined by the cache-line replacement policy. Common policies are first-in-first-out (FIFO), least-recently-used (LRU) and pseudo-least-recently-used (PLRU).

In a side-channel attack, an adversary extracts secret information from a target program via observations about side effects of the execution. For instance, if an implementation's memory accesses (and thereby the contents of the cache) depend on a secret, then the implementation might be vulnerable to a cache-side-channel attack. In this article, we consider three types of such attacks:

– *time*-based attacks such as [12, 16, 20] where the adversary observes the influence of caching via the total execution time.
– synchronous *access*-based attacks such as [17, 19, 26, 30], in which the adversary probes a shared memory cache after the program after termination using techniques such as Prime+Probe [19] or Flush+Reload [27].
– *trace*-based attacks such as [46, 18, 23], where the adversary deduces secret information from a trace of cache hits and misses. Such observations can be acquired for example based on the power consumption of the CPU [46].

3.2 Quantitative side-channel analysis via abstract interpretation

We consider a side channel to be a deterministic, discrete, memoryless channel from an input alphabet $\mathcal{X}$ to an output alphabet $\mathcal{O}$. $\mathcal{X}$ models all possible inputs of the target program, while $\mathcal{O}$ models the set of observations that are available to the adversary.

We model the probabilistic choice of the input by the discrete random variable X over $\mathcal{X}$, while the resulting side-channel observation is modeled by the random variable Obs over $\mathcal{O}$, which deterministically depends on X. For some $x \in \mathcal{X}$, we use $P(X = x)$ to denote the probability that the input was x. Furthermore, we denote the conditional probability that x was the input given some observation $obs \in \mathcal{O}$ with $P(X = x \mid Obs = obs)$.

Intuitively, side-channel leakage can be described as the reduction of uncertainty over the choice of the input given the side-channel output. We capture this using information-theoretic measures based on min-entropy, which were introduced by Smith [22].

We define the initial uncertainty over the choice of the input as the min-entropy of X, given by $H_\infty(X) = -\log_2 \max_{x\in\mathcal{X}} P(X = x)$. Next, we define the remaining uncertainty over the choice of the input given the side-channel observation as the conditional min-entropy of X given Obs, which is defined as $H_\infty(X \mid Obs) = -\log_2 \sum_{obs\in\mathcal{O}} P(Obs = obs) \cdot \max_{x\in\mathcal{X}} P(X = x \mid Obs = obs)$. Therefore, the leakage, i.e., the reduction of uncertainty, based on side-channel observations, is given by $H_\infty(X) - H_\infty(X \mid Obs)$.

Our approach to static quantitative side-channel analysis builds on the fact that $H_\infty(X) - H_\infty(X \mid Obs) \leq \log_2 |\mathcal{O}|$ holds under the assumption that X

is identically and independently distributed [22]. Since it might not be feasible to compute $\mathcal{O}$ for non-trivial programs, Köpf, Mauborgne, and Ochoa proposed an approach to derive an overaproximation of $\mathcal{O}$ for cache side channels using abstract interpretation [25].

The technique of abstract interpretation was introduced by Cousot and Cousot [1]. Given a model of execution with a set of states Σ and a concrete transfer function $next : \Sigma \to \Sigma$, an abstract interpretation consists of a join semi lattice of abstract states $(\Sigma^{\#}, \sqsubseteq, \sqcup)$ and an abstract transfer function $next^{\#} : \Sigma^{\#} \to \Sigma^{\#}$. One abstract state represents a set of concrete states according to the concretization function $\gamma : \Sigma^{\#} \to \mathcal{P}(\Sigma)$. If the abstract transfer function is locally sound (i.e. $\forall \sigma^{\#} \in \Sigma^{\#}. \forall \sigma \in \gamma(\sigma). next(\sigma) \in \gamma(next^{\#}(\sigma^{\#}))$) the fix point of the abstract transfer function obverapproximates all the states reachable by the concrete transfer function.

3.3 *CacheAudit*

In this article we build on *CacheAudit* [29] and its later extensions [33, 35, 44, 45]. This tool implements the approach presented in Section 3.2 to compute bounds on the cache-side-channel leakage of x86 programs.

To this end, the tool reasons about abstract states that model sets of execution snapshots of an x86 CPU. That is, each abstract state represents a set of possible snapshots, where each snapshot consists of a value-assignment for CPU flags and a value assignment for each register and memory location. For efficiency reasons, the abstract state is represented as a mapping from CPU flags to a mapping from register and memory locations to sets of values.

Moreover, an abstract state also models possible observations of multiple cache-based side-channel adversaries. *CacheAudit* considers two adversary models for synchronous *access*-based attacks, called `acc` and `accd`, as well as one adversary model for *trace*-based and one for *time*-based attacks called `trace` and `time`, respectively. For each of the adversary models, the overapproximated sets of observations derived from a set of abstract states are the following:

- $\mathcal{O}^{acc}$: The set of possible cache states after termination. A cache state maps each cached memory block to the corresponding cache line.
- $\mathcal{O}^{accd}$: The set of possible blinded cache states after termination. A blinded state maps every cache set to the number of cached memory blocks in this set.
- $\mathcal{O}^{trace} \subseteq \{hit, miss, none\}^*$: The set of possible traces of cache hits, cache misses, and steps without memory accesses that might have occurred during the execution of the target program.
- $\mathcal{O}^{time} \subseteq \mathbb{N}$: The set of execution times of the target program. The execution time is based on constants that model the duration of cache hits, misses, and other steps.

4 Increased precision for *CacheAudit*

We build on *CacheAudit* 0.3 to conduct our assessments. We previously published this version as part of the *RiCaSi* toolchain [44]. Although *CacheAudit* 0.3 delivered satisfactory results in prior studies, we are required to refine the underlying abstract transfer function to increase the precision for two x86 instructions. In this section we describe said modifications to the tool.

All implementations we investigate throughout this publication use the SHL or SHR instruction on 8 bit operands. However, the abstract transfer function of *CacheAudit* 0.3 only implements a coarse overapproximation of these instructions. This imprecision prevented us from obtaining sufficiently precise results for the analysis.

According to the Intel Software Developer Manual [31], the x86 instructions SHL and SHR perform a logical left and right shift, respectively. The instructions shift the value of a 32 or 8 bit operand in-place by a second operand that is given either in the *CL* register or as an 8 bit immediate value.

We extend the abstract transfer function for the case in which both operands are represented in the set abstract domain. In this article, we define the abstract semantics of both instructions as functions that map sets of values of both operands to sets of the resulting values (denoted by $SHL_8^{\#} : \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$ and $SHR_8^{\#} : \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$, respectively). We define the abstract semantics of SHL and SHR as follows:

$$SHL_8^{\#}(OP1, OP2) := \{(op1 << op2) \mod 8 \mid op1 \in OP1 \wedge op2 \in OP2\}$$
$$SHR_8^{\#}(OP1, OP2) := \{(op1 >> op2) \mod 8 \mid op1 \in OP1 \wedge op2 \in OP2\}$$

We argue that these abstract semantics are sound. This follows from the fact that the result contains all possible shifts of the values from the set representing the first operands by the values from the set representing the second operand.

For the actual implementation, the functions are integrated into the existing abstract transfer function. We do not consider a precise overapproximation for the flags and instead map to an abstract state in which the value for each flag is unknown after interpreting the 8 bit variant of the SHL or SHR instruction. Thus, local consistency of the new abstract transfer function follows.

The refined abstract transfer function is now precise enough for our analyses. Reason being that the overapproximated resulting set of values is as precise as possible given the set of input values. Moreover, the course overapproximation of the flags has no negative effect for the considered implementations. This is because in none of the implementations, SHL or SHR instructions are placed such that they potentially influence the control flow or such that the flags resulting from the instructions are read into a register.

5 Analysis of AES from *OpenSSL*, *mbedTLS* and *Nettle*

As a reference point for our analysis across modes of operation, we first establish a base-line result for AES-128 from *OpenSSL*, *mbedTLS* and *Nettle*. To this end, we build on an assessment of AES from our prior work [35]. Since then, the crypto libraries, the compiler and the analysis tool have been updated. We first revisit the analyses from [35]. Then we investigate the effect of each update and assess their impact on the leakage bounds.

5.1 Base–line analyses

In [35], we investigated AES-128 implementations from *OpenSSL* 1.0.1.t, *mbedTLS* 2.2.1 and *Nettle* 3.2 compiled with *gcc* 4.8.3 using *CacheAudit* 0.2b. The leakage bounds computed in [35] are presented in Figure 4 for a 128 KiB, 4-way associative cache with a line size of 64 B and the FIFO replacement strategy.

In [35], we found that the leakage bounds for the *OpenSSL* implementation are smaller when compared to the *mbedTLS* and *Nettle* implementation. Based on manual code review and a cross cache-size analysis, we were able to attribute this difference to the different implementation strategies for the last AES round. While the *OpenSSL* implementation reuses existing tables $T_0 - T_3$ to perform the last round, the *mbedTLS* and *Nettle* implementation use an additional S-Box for the same task. Accesses to the additional lookup table have the potential to carry more information that might be observable through a cache side channel, which explains the higher leakage bounds.

Library	Attacker Model			
	`acc`	`accd`	`trace`	`time`
OpenSSL 1.0.1.t	64.0	64.0	196.0	7.7
mbedTLS 2.2.1	69.0	69.0	199.0	7.7
Nettle 3.2	69.0	69.0	199.0	7.7

Fig. 4: Leakage bounds [bit], AES-128 old setup, 128 KiB 4-way FIFO.

Since [35], *OpenSSL* 1.1.1d [47], *mbedTLS* 2.16.5 [49] and *Nettle* 3.5 [51] have been released. The updates to *OpenSSL* and *Nettle* did not affect the source code of the analyzed AES-128 implementation, while the updated version of AES-128 from *mbedTLS* contains a new feature. Since version 2.16.4, *mbedTLS* now also zeroizes local variables to lessen the impact of buffer-overread vulnerabilities [48].

Even for the implementations that have not been changed, updating *gcc* from version 4.8.3 to 9.3.0 yields different binaries. But the differences in the binaries are only minor and operations that potentially impact the side-channel leakage (e.g. `mov` instructions that dereference secret-dependent pointers) are not affected. For the example of the *OpenSSL* binaries the old binary consists of 485 instructions while the new binary consists of 497 instructions. Notable differences are the start addresses of the functions, the use of different jump instructions (jne vs je followed by jmp), using xor instead of or in two instances, and the old binary using lea instructions that are replaced by mov operations in the new binary. The types of differences for the new *mbedTLS* and *Nettle* binaries are similar.

Since [35], we have also updated our static analysis tool. To analyze lattice based crypto, we have improved the precision for multiple instructions in [33], we extended the tool to handle very large binaries by implementing the code that generates the CFG in a tail-recursive style in [44] and we extended the abstract transfer function to cover the x87 FPU in [45]. Furthermore, we refined the abstract transfer function for the 8 bit variants of SHL and SHR in Section 4.

Library	Attacker Model			
	`acc`	`accd`	`trace`	`time`
OpenSSL 1.1.1d	64.0	64.0	196.0	7.7
mbedTLS 2.16.5	69.0	69.0	199.0	7.7
Nettle 3.5	69.0	69.0	199.0	7.7

Fig. 5: Leakage bounds [bit], AES-128 updated setup, 128 KiB 4-way FIFO.

The leakage bounds we obtained for the updated libraries, updated compiler and updated analysis tool are presented in Figure 5. We find that the leakage bounds are exactly the same and were not affected by any of the updates. This gives us confidence that the update to *mbedTLS* and the variations in the generated x86 code introduced by the new compiler do not introduce additional potential for leakage. Moreover, these results show that our findings from [35] hold for the updated setup.

6 Analysis across CBC mode, CTR mode and CCM

In this section we investigate *OpenSSL* AES-128 in CBC mode, CTR mode and CCM. We start by presenting leakage bounds for the encryption of two-block messages with the *OpenSSL* implementation and combine this with a manual code analysis to justify our findings. Then we compare the implementation from *OpenSSL* to the implementations from *mbedTLS* and *Nettle*. In the end, we show that our findings are also applicable to messages consisting of more than two blocks.

6.1 Cache-side-channel analysis of *OpenSSL* CBC, CTR, and CCM

For the comparison, we initially consider the encryption of a two-block message (i.e. a 256 bit plaintext) with AES-128 in the modes of operations in question. We configure CCM with a 96 bit nonce, while CBC and CTR mode both take an IV with a fixed length of 128 bit. We configure our static analysis such that all input parameters except the plaintext length are secret. Thus, for the analysis of CBC and CTR, we have 512 bit and for CCM we have 480 bit of secret input.

We make minor changes to the source code of *OpenSSL* prior to compilation to prepare the binary for the analysis. All modes are parameterized over the block cipher using function pointers, which generate indirect jump instructions that are not supported by *CacheAudit*. Since we are only interested in the instantiation of each mode with AES, we simply replace all occurrences of function pointers with direct calls to AES. Furthermore, we replace the functions *memset* and *memcpy* to an implementation from the *musl* C standard library[50] instead of their *gLibc* counterpart. The reason is that the abstract transfer function of *CacheAudit* is imprecise for multiple instructions present in the *memset* and *memcpy* x86 code from *gLibc*.

We use *gcc* 9.3.0 to compile one x86 binary for each mode of operation. The binaries first call the function for the AES key expansion `AES_set_encrypt_key`, then call some auxiliary functions to set up some data structures and finally invoke the mode-specific encryption function `CRYPTO_{cbc,ctr,ccm}128_encrypt`. We check the correctness of the analyzed binaries with test vectors provided by NIST [10, 14].

Mode	input size	Attacker Model			
		`acc`	`accd`	`trace`	`time`
CBC	512.0	64.0	64.0	356.0	8.5
CTR	512.0	64.0	64.0	356.0	8.5
CCM	480.0	64.0	64.0	996.0	10.0

Fig. 6: Leakage bounds [bit], *OpenSSL*, 128 KiB 4-way FIFO

The analysis results for 128 KiB, 4-way associative caches with a line size of 64 B and FIFO replacement strategy are presented in Figure 6. Our key findings are that for CBC and CTR we obtain exactly the same leakage bounds, while the analysis of CCM results in higher `trace` and `time` bounds. Furthermore, we have that the leakage bounds for both covered access-based adversary models are the same as for the base-line analysis of AES-128 shown in Figure 5.

We interpret these results via a manual code review. Excerpts of the *OpenSSL* implementation of CBC mode and CTR mode are presented in Listing 1.1 and 1.2. The library provides a straight forward implementation of CBC and CTR as it was introduced in Section 2.2. For both implementations, the input is stored at `in`, the output is stored at `out`, the length of the plaintext is stored in `len`, the key is stored at `key` and the IV is stored at `iv` or `ivec`. Moreover, in CTR, the variable `ecount_buf` points to the round-specific one-time pad.

Listing 1.1: *OpenSSL* 1.1.1d CBC Mode

```c
void CRYPTO_cbc128_encrypt(/* ... */){
   /* ... */
   while (len >= 16) {
      for (n = 0; n < 16; ++n)
        out[n] = in[n] ^ iv[n];

      AES_encrypt(out, out, key);
      iv = out; len -= 16; in += 16; out += 16;
    }
   /* ... */
}
```

Listing 1.2: *OpenSSL* 1.1.1d CTR Mode

```c
static void ctr128_inc(unsigned char *counter) {
  u32 n = 16, c = 1;

  do {
      --n; c += counter[n];
      counter[n] = (u8)c; c >>= 8;
  } while (n);
}

void CRYPTO_cbc128_encrypt(/* ... */){
   /* ... */
    while (len >= 16) {
        AES_encrypt(ivec, ecount_buf, key);

        ctr128_inc_aligned(ivec);

        for (n = 0; n < 16; n += sizeof(size_t))
          *(size_t *)(out + n) =
              *(size_t *)(in + n) ^ *(size_t *)(ecount_buf + n);

        len -= 16; out += 16; in += 16; n = 0;
      }
   /* ... */
}
```

Note that, both implementations only contain memory accesses with statically known offsets. This includes all accesses to `in`, `out`, `iv` or `ivec`, `ecount_buf` and the counter value `counter`. Moreover, the number of loop iterations is statically known in all cases (recall that we assume a fixed message length). From this we conclude that there are no leaks except for the invocations of AES.

An excerpt of the *OpenSSL* implementation of CCM is shown in Listing 1.3. Again, the code is a straight-forward implementation of CCM as is introduced in Section 2.2. The plaintext is stored at `inp`, the ciphertext is stored at `out`, the length of the plaintext is stored in `len`, the key is stored at `key`, the nonce is stored at `ctx→nonce`, the current MAC is stored at `ctx→cmac` and the round-specific one-time pad is stored at `scratch`. Interestingly, the xor operation on

128 bit values is implemented via applying the xor operation on the lower and upper 64 bit separately instead of a Byte-wise operation as in the CBC and CTR mode implementation.

Listing 1.3: *OpenSSL* 1.1.1d CCM

```c
unsigned int n = 8; u8 c;

    counter += 8; do { --n; c = counter[n];
        ++c; counter[n] = c;
        if (c) return;
    } while (n);
}

void CRYPTO_ccm128_encrypt(/* ... */){
    /* ... */
     while (len >= 16) {

        ctx->cmac.u[0] ^= ((u64 *)inp)[0];
        ctx->cmac.u[1] ^= ((u64 *)inp)[1];

        AES_encrypt(ctx->cmac.c, ctx->cmac.c, key);
        AES_encrypt(ctx->nonce.c, scratch.c, key);

        ctr64_inc(ctx->nonce.c);

        ((u64 *)out)[0] = scratch.u[0] ^ ((u64 *)inp)[0];
        ((u64 *)out)[1] = scratch.u[1] ^ ((u64 *)inp)[1];

        inp += 16; out += 16; len -= 16;
    }
    /* ... */
    for (i = 15 - L; i < 16; ++i) ctx->nonce.c[i] = 0;

    AES_encrypt(ctx->nonce.c, scratch.c, key);
    ctx->cmac.u[0] ^= scratch.u[0]; ctx->cmac.u[1] ^= scratch.u[1];
    /* ... */
}
```

We again find that all memory accesses have statically known offsets. This includes all accesses to `inp`, `out`, `ctx`→`nonce`, `ctx`→`cmac` and `scratch`. Therefore, we conclude that there is no additional leakage towards our access based adversaries.

Notably, we have that the running time of the counter-increment function (ctr64_inc) actually depends on the current counter value. This function increments the 64 bit value in a Byte-wise fashion, starting at the lowest Byte. The next Byte only has to be incremented if incrementing the current Byte results in an overflow. But this does not lead to any leakage. The reason is that the part of the counter value that is incremented is always initialized with 0 independent of the nonce. Thus, the running time of the encryption remains static for a fixed message length.

Since we did not find any secret-dependent memory accesses or conditionals, we conclude that there are no code sections in the CBC mode, CTR mode and CCM implementation from *OpenSSL* that induce any additional cache-side-channel leakage. Hence, the leakage bounds presented in Figure 6 are exclusively influenced by invocations of AES.

To summarize, we find that we can attribute similar security guarantees with respect to cache side channels across the *OpenSSL* implementations of CBC mode and CTR mode. Furthermore, we show that the leakage bounds for CCM are higher when compared to CBC and CTR mode due to the fact that CCM applies twice as many AES encryptions. Thus, the additional authentication functionality of CCM comes at the cost of worse security guarantees regarding the trace and timing adversaries.

6.2 Comparison across *OpenSSL*, *mbedTLS* and *Nettle*

We also investigate how the differences in the implementation of AES-128 we studied in prior work [35] and revisited in Section 5 affect the leakage bounds when AES is applied in a mode of operation. Moreover, we also explore if there are any design choices in the CBC, CTR and CCM implementations from *mbedTLS* 2.16.5 and *Nettle* 3.5 that influence the leakage bounds.

For the analysis of the implementations from *mbedTLS* 2.16.5 and *Nettle* 3.5, we apply similar code transformations and mirror the setup for the binaries and *CacheAudit* from the previous section. For *mbedTLS*, we investigate binaries that first call the respective key setup function `mbedtls_aes_setkey_enc` in the case of CBC or CTR mode or `mbedtls_ccm_setkey` for CCM and then the encryption function `mbedtls_aes_crypt_{cbc|ctr}` or `mbedtls_ccm_encrypt_and_tag`, respectively. For *Nettle*, we analyze binaries that first call the respective key setup function `aes128_set_encrypt_key` or `ccm_aes128_set_key` and then call the encryption function `{CBC|CTR}_ENCRYPT_128` or `ccm_aes128_encrypt_message`. The results of the analyses are presented in Figure 7.

Mode	Attacker Model				
	input size	`acc`	`accd`	`trace`	`time`
CBC	512.0	69.0	69.0	359.0	8.5
CTR	512.0	69.0	69.0	367.2	12.9
CCM	480.0	69.0	69.0	999.0	10.0

(a) *mbedTLS*

Mode	Attacker Model				
	input size	`acc`	`accd`	`trace`	`time`
CBC	512.0	69.0	69.0	359.0	8.5
CTR	512.0	69.0	69.0	367.2	12.9
CCM	480.0	69.0	69.0	999.0	10.0

(b) *Nettle*

Fig. 7: Leakage bounds [bit], across libraries, 128 KiB 4-way FIFO

Our observations are threefold. (1) We again have that the leakage bounds for synchronous access-based adversaries are the same as for AES-128 for all modes of operation. (2) We have that the time and trace leakage bounds are higher for CTR mode and significantly higher for CCM. (3) The differences in leakage bounds for all covered modes across the three libraries are similar to the differences we saw in our study of the underlying AES-128 implementations.

Concerning observation (2), we are able to manually trace this increase of the timing bounds to one section of the code. We find for both libraries that they leak the value of the IV due to a branching in the counter-increment function which we considered secret for the analysis. However, since the IV is is a public value [9], we determine that this leak can be dismissed.

Together with observation (1) and (3), we are able to conclude that the differences in the implementation of the last AES round are neither masked nor significantly amplified by applying AES in a mode of operation. In our code inspection, we also found no additional potential leaks in the *mbedTLS* and *Nettle* implementations of CBC, CTR and CCM.

6.3 Investigating longer block messages

Both, in Section 6.2 and Section 6.2 we found that across all modes and libraries, the leakage bounds with respect to our synchronous access-based adversary models did not change when compared to the leakage bounds of the underlying AES implementation. Furthermore, we discovered that for CCM, a mode that applies twice as many AES encryptions, the timing leakage bounds were slightly and the trace leakage bounds were significantly higher when compared to the other two modes. These results suggest that the leakage bounds scale differently with the number of AES encryptions depending on the adversary model.

To test this hypothesis and to further investigate the leakage-bound behaviour, we broaden our assessment towards longer messages that require more AES encryptions to encrypt. Since the implementations from all investigated libraries exhibit the same behaviour regarding our hypothesis, we again focus on *OpenSSL*. We expand the scope to the encryption of up to 32 message blocks i.e. 4096 bit long messages. The number of blocks we are able to analyze is limited by the speed of our static analysis. Considering that the limit on the plain text length of one record in TLS 1.3 is roughly 2^{17} bit [39] , we do not cover the full spectrum of possible use cases, but our results show clear trends of the leakage bounds wrt. increasing message block count.

The results across all modes and adversary models are shown in Figure 8. We provide one graph for each adversary model, with one line for each mode of operation. On the x-axis, we show the leakage bounds, while on the y-axis we show the message length. To visualize trends we have connected the leakage bounds of successive message lengths.

We find that the access-based leakage bounds do not increase with the number of blocks, while the timing-based bounds increase logarithmcally and the trace-based bounds increase linearly with the message length. Moreover, we have that the timing and trace leakage bounds for CCM increase faster when compared to CBC and CTR mode.

We interpret the fact that the leakage bounds for both access-based adversary models stay unchanged for even longer messages as follows. Both of our

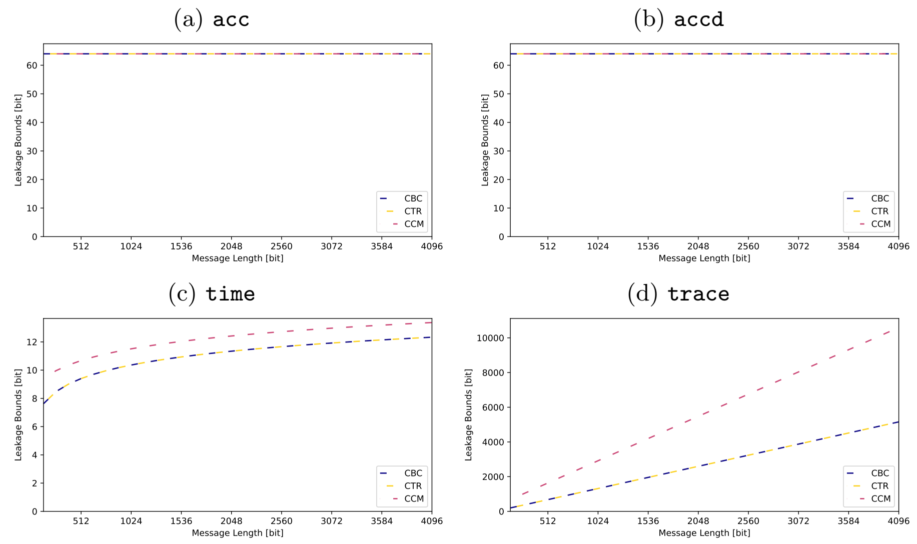


Fig. 8: *OpenSSL* CBC, CTR and CCM, 128 KiB 4-way FIFO, up to 32 blocks.

access-based adversary models are synchronous (i.e. they take one observation after execution of the target program). Thus, both covered access-based adversaries can not distinguish whether a lookup-table entry was cached during the first or during any subsequent AES encryption. Therefore the upper bound on the information that a synchronous access-based adversary can learn does not increase with additional AES invocations.

This argument, however, does not hold for our trace and time-based adversary models. Regarding timing attacks, subsequent invocations of the AES encryption still lead to more variations in the execution time depending on which lookup-table entries have been cached. The logarithmic scaling suggests that the number of new possible timings and therefore the upper bound on the leaked information decreases with each new AES encryption.

Regarding trace-based attacks, each encryption extends the trace of cache-hits and misses that encode information on the key and the next message block (recall that AES potentially leaks both the key and the message). Thus, we have that multiple AES encryptions do not hide any information leaked by individual AES encryptions, which explains why the leakage bounds increase linearly.

These results do not only confirm our hypothesis regarding the scaling, but we observe clear trends for each adversary model. Therefore, we deduce that the observations made in Section 6.2 and Section 6.2 hold for longer messages that go beyond the analyzed scenarios.

7 Related Work

In the following section, we provide an overview on related work. We describe different static side-channel analysis tools and compare them to our version of *CacheAudit*. Then we cover prominent cache-side-channel attacks and studies on AES and the modes of operation to show how they relate to our observed leakage bounds.

7.1 Static side-channel analysis tools

We build on *CacheAudit* 0.3, which is part of a toolchain for cache-side-channel mitigation using circuit compilation, called *RiCaSi* [44]. *CacheAudit* was originally published by Doychev, Köpf, Mauborgne, and Reineke [29]. We extended the tool in [35] for the analysis of different AES implementations, in [33] for the analysis of lattice-based cryptography, and in [45] for the analysis of software used for quantum key distribution. In the meantime, Doychev and Köpf [34] extended *CacheAudit* to study different modular exponentiation algorithms.

A different quantitative cache-side-channel-analysis tool is *CHALICE* [41]. This tool applies a combination of symbolic execution and testing to quantify the leakage regarding different cache-side-channel adversaries. Moreover, there also exist multiple quantitative approaches based on symbolic execution and model counting for the analysis of Java programs [32, 36].

Lastly, we also mention qualitative tools that focus on the detection of cache-side-channels in code. *CacheS* [43] is similar to *CacheAudit* in that abstract interpretation is performed as a reachability analysis. They gain analysis speed by coarsely overapproximating values that have no impact on the side-channel leakage. *CacheD* [37] uses symbolic execution to find execution traces with different memory behavior to identify insecure code. *DATA* [40] compares so-called address traces, which are recorded during executions of the target program to detect cache-side-channel leakage. None of these tools would have been suitable for the study in this article, since they do not provide bounds on the potential leakage of the analyzed program.

7.2 Cache-side-channels of AES-128 and modes of operation

Conceptually, our work is closest to our prior work in [35]. We quantified the cache-side-channel leakage of various lookup-table-based AES-128 implementations for older versions of the same libraries and compiler. In section Section 5, we showed that our previous results are still valid for the new setup and extend our study to AES-128 in CBC and CTR mode, and CCM in Section 6.

As far as we know, there are no published cache-side-channel attacks that specifically target AES in any of the covered modes of operation yet. Lapid and Wool [38] attacked AES-256 in Galois Counter Mode (GCM), which we did not cover in this article. They targeted the AES lookup table via Prime+Probe and were able to extract the secret key with 40 min of measurements and 30 min of analysis. The attack did not benefit from the encryption of multiple blocks. This is in line with our results that the upper leakage bounds for our access-based adversary models did not increase for the encryption of longer messages.

There are many related attacks that focus on plain AES-128. For example, the timing-based attack by Bernstein [16] and the trace-based attack by Acıiçmez and Koç [18] both exploit characteristics of the lookup-table-based AES software implementation from *OpenSSL*. The observational capabilities of such attackers are modeled in *CacheAudit* by the `time` and `trace` adversary models, respectively. Furthermore, there are asynchronous access-based attacks presented by Gullasch, Bangerter, and Krenn [24] and Yarom and Falkner [27]. In *CacheAudit*, the observations of a synchronous variant of an access-based attacker are captured by the adversary model `acc`.

While there are many hardening techniques for software implementations of AES such as, but not limited to, preloading [11], using smaller tables or avoiding memory accesses [19], or randomizing the control flow of the binary [28], we only know of one technique specifically tailored to a mode of operation. Käsper and Schwabe [21] propose a bitsliced implementation of AES-128-GCM, which essentially simulates a hardware implementation in software in such a way that it protects against cache-side-channel attacks.

8 Conclusion

AES is widely considered secure and used in practice, but it is well known that many software implementations of AES are vulnerable to cache side channels. For this article, we studied AES implementations from three libraries: *OpenSSL* 1.1.1d, *mbedTLS* 2.16.5, and *Nettle* 3.5. For each of these implementations, we derived upper bounds on the information leakage via cache side channels. We computed these leakage bounds by applying an established combination of abstract interpretation and information theory. To our knowledge, this is the first time these versions of the implementations were analyzed wrt. side channels (prior versions were analyzed in [35]). The main technical novelty of this article is the coverage of AES in three modes of operation: CBC, CTR, and CCM.

For AES-128 we obtained non-zero leakage bounds for all implementations. For *OpenSSL* we derived upper leakage bounds of 64 bits (access-based adversaries), 196 bits (trace-based adversary), and 7.7 bits (timing-based adversary). For *mbedTLS* we derived upper leakage bounds of 69 bits (access-based adversaries), 199 bits (trace-based adversary), and 7.7 bits (timing-based adversary). The leakage bounds for the AES-128 implementation from *Nettle* are equal to the bounds for the AES-128 implementation from *mbedTLS*. We argued that the usage of an additional 256 B lookup-table in the *mbedTLS* and *Nettle* implementations is most likely the reason for the higher leakage bounds when considering access-based and trace-based adversaries.

When analyzing the encryption of two message blocks, we found that the leakage bounds for access-based adversaries for the *OpenSSL* AES-128 implementation in CBC, CTR, and CCM are identical to the bounds for the encryption of a single message block. However, we obtained different leakage bounds for timing-based and trace-based adversaries. More concretely, we derived a leakage bounds of 8.5 bits for CBC and CTR, and we derived a leakage bound of 10 bits for CCM for a timing-based adversary. For a trace-based adversary, we computed leakage bounds of 356 bits (for CBC and CTR) and 996 bits (for CCM). For the encryption of two message blocks with *mbedTLS* and *Nettle* AES-128 in CBC, CTR and CCM, we obtained the same bounds as for the encryption of a single message block when considering access-based adversaries. Moreover, we computed upper leakage bounds of 359.0 bits (CBC), 367.2 bits (CTR), and 999.0 bits (CCM) when considering trace-based adversaries. That is, there is a slight increase of the difference between the leakage bounds for the AES implementations from *OpenSSL* and the other libraries when encrypting multiple message blocks using the same mode of operation, but only for CTR and CCM.

In summary, we obtained smaller upper leakage bounds for the AES-128 implementation from OpenSSL than for the implementations from the other two libraries. Encrypting multiple message blocks did not alter the leakage bounds when considering access-based attackers. However, the leakage bounds increased when considering trace-based and timing-based attackers. Finally, which mode of operation was used (CBC, CTR, or CCM) had an effect on the leakage bounds, and the difference was most significant when considering trace-based attackers.

Acknowledgements Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1119 - 236615297.

References

[1] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, pp. 238–252.

[2] National Bureau of Standards. *Data Encryption Standard (DES)*. Federal Information Processing Standards Publication 46. 1977.

[3] Whitfield Diffie and Martin E Hellman. “Privacy and authentication: An introduction to cryptography”. In: *Proceedings of the IEEE* 67.3 (1979), pp. 397–427.

[4] National Institute of Standards and Technology. *DES Modes of Operation*. Federal Information Processing Standards Publication 81. 1980.

[5] Paul Kocher. “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems”. In: *Advances in Cryptology — CRYPTO ’96*. 1996, pp. 104 –113.

[6] Joan Daemen and Vincent Rijmen. *AES proposal: Rijndael*. 1999.

[7] Paul Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis”. In: *Advances in Cryptology — CRYPTO ’99*. 1999, pp. 388–397.

[8] Joan Daemen and Vincent Rijmen. *Specification for the advanced encryption standard (AES)*. Federal Information Processing Standards Publication 197. 2001.

[9] Morris Dworkin. *Recommendation for block cipher modes of operation. methods and techniques*. Special Publication (NIST SP) - 800-38A. 2001.

[10] Lawrence E Bassham III. “The advanced encryption standard algorithm validation suite (AESAVS)”. In: *NIST Information Technology Laboratory* (2002).

[11] Daniel Page. “Defending against cache-based side-channel attacks”. In: *Information Security Technical Report* 8.1 (2003), pp. 30–44.

[12] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. “Cryptanalysis of DES implemented on computers with cache”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. 2003, pp. 62–76.

[13] D Whiting, R Housley, and N Ferguson. *Counter with CBC-MAC (CCM)*. RFC 3610. 2003.

[14] Lawrence E Bassham III. “The CCM Validation System (CCMVS)”. In: *NIST Information Technology Laboratory* (2004).

[15] Morris J Dworkin. *Recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality*. Special Publication (NIST SP) - 800-38C. 2004.

[16] Daniel J. Bernstein. *Cache-timing attacks on AES*. Tech. rep. University of Illinois at Chicago, 2005.

[17] Colin Percival. “Cache missing for fun and profit”. In: *Proceedings of BSDCan*. 2005.

[18] Onur Acıiçmez and Çetin Kaya Koç. “Trace-driven cache attacks on AES (short paper)”. In: *International Conference on Information and Communications Security*. 2006, pp. 112–121.

[19] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache attacks and countermeasures: the case of AES”. In: *Cryptographers' track at the RSA conference*. 2006, pp. 1–20.

[20] Onur Acıiçmez, Werner Schindler, and Çetin K Koç. “Cache based remote timing attack on the AES”. In: *Cryptographers' track at the RSA conference*. 2007, pp. 271–286.

[21] Emilia Käsper and Peter Schwabe. “Faster and timing-attack resistant AES-GCM”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2009, pp. 1–17.

[22] Geoffrey Smith. “On the foundations of quantitative information flow”. In: *International Conference on Foundations of Software Science and Computational Structures*. 2009, pp. 288–302.

[23] Xin-Jie Zhao and Tao Wang. “Improved Cache Trace Attack on AES and CLEFIA by Considering Cache Miss and S-box Misalignment.” In: *IACR Cryptol. ePrint Arch.* (2010), p. 56.

[24] David Gullasch, Endre Bangerter, and Stephan Krenn. “Cache games–bringing access-based cache attacks on AES to practice”. In: *2011 IEEE Symposium on Security and Privacy*. 2011, pp. 490–505.

[25] Boris Köpf, Laurent Mauborgne, and Martin Ochoa. "Automatic Quantification of Cache Side-Channels". In: *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*. 2012, pp. 564–580.

[26] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a minute! A fast, Cross-VM attack on AES". In: *International Workshop on Recent Advances in Intrusion Detection*. 2014, pp. 299–319.

[27] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security Symposium*. 2014, pp. 719–732.

[28] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity." In: *NDSS*. 2015, pp. 8–11.

[29] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. "Cacheaudit: A tool for the static analysis of cache side channels". In: *ACM Transactions on Information and System Security (TISSEC)* 18.1 (2015), pp. 1–32.

[30] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "S $ A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES". In: *Symposium on Security and Privacy*. 2015, pp. 591–604.

[31] *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D*. October 2016 (updated November 2019).

[32] Corina S Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. "Multi-run side-channel analysis using Symbolic Execution and Max-SMT". In: *Proceedings of the 29th Computer Security Foundations Symposium (CSF)*. 2016, pp. 387–400.

[33] Nina Bindel, Johannes Buchmann, Juliane Krämer, Heiko Mantel, Johannes Schickel, and Alexandra Weber. "Bounding the cache-side-channel leakage of lattice-based signature schemes using program semantics". In: *Proceedings of the 10th International Symposium on Foundations & Practice of Security (FPS)*. 2017, pp. 225–241.

[34] Goran Doychev and Boris Köpf. "Rigorous Analysis of Software Countermeasures against Cache Attacks". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. 2017, pp. 406–421.

[35] Heiko Mantel, Alexandra Weber, and Boris Köpf. "A Systematic Study of Cache Side Channels across AES Implementations". In: *International Symposium on Engineering Secure Software and Systems*. 2017, pp. 213–230.

[36] Quoc-Sang Phan, Lucas Bang, Corina S Pasareanu, Pasquale Malacaria, and Tevfik Bultan. "Synthesis of Adaptive Side-Channel Attacks". In: *Proceedings of the 30th Computer Security Foundations Symposium (CSF)*. 2017, pp. 328–342.

[37] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. "Cached: Identifying cache-based timing channels in production software". In: *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 2017, pp. 235–252.

[38] Ben Lapid and Avishai Wool. "Cache-attacks on the ARM TrustZone implementations of AES-256 and AES-256-GCM via GPU-based analysis". In: *International Conference on Selected Areas in Cryptography*. 2018, pp. 235–256.

[39] Eric Rescorla and Tim Dierks. *The transport layer security (TLS) protocol version 1.3*. RFC 8446. 2018.

[40] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. "{DATA}–Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 603–620.

[41] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. "Quantifying the information leakage in cache attacks via symbolic execution". In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.1 (2019), pp. 1–27.

[42] Kerry McKay and David Cooper. *Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations*. Special Publication (NIST SP) - 800-52 Rev. 2. 2019.

[43] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. "Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation". In: *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. 2019, pp. 657–674.

[44] Heiko Mantel, Lukas Scheidel, Thomas Schneider, Alexandra Weber, Christian Weinert, and Tim Weißmantel. "RiCaSi: Rigorous Cache Side Channel Mitigation via Circuit Compilation". In: *International Conference on Cryptology and Network Security*. 2020, pp. 505–525.

[45] Alexandra Weber, Oleg Nikiforov, Alexander Sauer, Johannes Schickel, Gernot Alber, Heiko Mantel, and Thomas Walther. "Cache-Side-Channel Quantification and Mitigation for Quantum Cryptography". In: *European Symposium on Research in Computer Security*. 2021, pp. 235–256.

[46] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo. In: *Proceedings. ITCC 2005 International Conference on Information Technology: Coding and Computing*.

[47] OpenSSL Software Foundation. *OpenSSL (Version 1.1.1d)*. `https://www.openssl.org/source/openssl-1.1.1d.tar.gz`. [Online; accessed Nov-2021].

[48] ARM Limited. *mbedTLS 2.16.4 Patchnotes*. `https://tls.mbed.org/tech-updates/releases/mbedtls-2.16.4-and-2.7.13-released`. [Online; accessed Nov-2021].

[49] ARM Limited. *mbedTLS (Version 2.16.5)*. `https://tls.mbed.org/download/start/mbedtls-2.16.5-apache.tgz`. [Online; accessed Nov-2021].

[50] *musl Home Page*. `https://musl.libc.org/`. [Online; accessed Nov-2021].

[51] Niels Möller. *Nettle (Version 3.5)*. `https://ftp.gnu.org/gnu/nettle/nettle-3.5.tar.gz`. [Online; accessed Nov-2021].