Cache vs. Key-Dependency: Side Channeling an Implementation of Pilsung

Daniel Genkin[1], Romain Poussier[2], Rui Qi Sim[3], Yuval Yarom[3,4] and Yuanjing Zhao[3]

[1] University of Michigan
[2] Nanyang Technological University
[3] University of Adelaide
[4] Data61

Abstract. Over the past two decades, cache attacks have been identified as a threat to the security of cipher implementations. These attacks recover secret information by combining observations of the victim cache accesses with the knowledge of the internal structure of the cipher. So far, cache attacks have been applied to ciphers that have fixed state transformations, leaving open the question of whether using secret, key-dependent transformations enhances the security against such attacks.
In this paper we investigate this question. We look at an implementation of the North Korean cipher Pilsung, as reverse-engineered by Kryptos Logic. Like AES, Pilsung is a permutation-substitution cipher, but unlike AES, both the substitution and the permutation steps in Pilsung depend on the key, and are not known to the attacker. We analyze Pilsung and design a cache-based attack. We improve the state of the art by developing techniques for reversing secret-dependent transformations. Our attack, which requires an average of eight minutes on a typical laptop computer, demonstrates that secret transformations do not necessarily protect ciphers against side channel attacks.

Keywords: Prime+Probe · Cache Attack · Side-Channel · Key-Dependent Cipher · Misaligned Tables

1 Introduction

The seminal work of Kocher [Koc96], demonstrated the need to protect cryptographic primitives against implementation attacks. *Side channel* attacks leak information from the implementation of a cryptographic primitive through its interaction with the environment. These attacks are often divided into two main categories. *Physical* attacks leak the information through the interaction with the physical environment, e.g. via variations in power consumption [KJJ98], electromagnetic emanations [AARR02; GMO01; QS01], and photonic [SNK+13] or acoustic [GST14] emissions. Conversely, *microarchitectural* channels [GYCH18], leak the information of a software implementation via its shared use of processor components. While many microarchitectural channels exist [CABH+19; GRBG18; IMB+19b; PGM+16], arguably the most exploited microarchitectural component is the processor cache. Indeed, since the first attacks [OST06; Per05; TSS+03; TTMH02], multiple attack techniques have been developed, exploiting leaks from implementations of symmetric ciphers [IAES15; IAIES14; OST06; SP13; TSS+03; TTMH02; ZW10], pre- and post-quantum public key systems [BBG+17; BH09; GBK11; GPTY18; GVY17; PBY17; Per05; PGBY16; YF14; ZJRR12], and other cryptographic primitives [DDME+18].

To maintain security, block ciphers must use non-linear components such as *substitution boxes* (S-Boxes) and repeatedly apply them to a secret intermediate state across all the

Licensed under Creative Commons License CC-BY 4.0.
IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925, Vol. 2020, No. 1, pp. 231–255
DOI:10.13154/tches.v2020.i1.231-255

cipher's rounds. However, the non-linear nature of these components also means that they are typically implemented using tables (either holding S-Boxes or their optimized and precomputed version, such as T-Tables for the AES) which are then accessed in a way that highly depends on the cipher's internal state. From a side channel perspective, the implications of such an implementation are disastrous: an attacker monitoring the cache can deduce the target's memory access patterns, which directly reveal the cipher's state. From this state, the attacker can simply reverse the cipher, deducing the key.

While the attack outlined above works in theory, in practice it is typically impossible to completely deduce the cipher's state by merely observing the cache after a single table application. The main limitation is that the cache stores information in *cache lines*, which are larger than the table entries. As an access somewhere in the line results in the CPU caching the entire line, the attacker is unable to distinguish nearby table entries. This results in the attacker recovering only partial information about the cipher's internal state during a single round, which is insufficient for key recovery. Consequently, most cache attacks typically recover information from multiple rounds, and subsequently achieve key extraction by combining it. More specifically, the attacker makes a hypothesis on some missing key bits, and then subsequently uses the cipher's structure to determine the output of the cipher's first round. The output of the first round is then combined with side channel leakage from the cipher's second round to obtain key extraction. Such an approach, however, inherently relies on the attacker's ability to predict the cipher's data propagation. While this is the case for popular ciphers, such as AES, which have a fixed key-independent structure, much less is known about side channel leakage from ciphers whose basic structures (e.g. S-Boxes and other transformations) are also key dependent.

For power and electromagnetic analysis, a negative indication about the protection provided by key-dependent transformations was recently given [GOC16; MCS17]. However, to the best of our knowledge, no such investigation has ever been undertaken with microarchitectural side channels. As an example of a cipher implementation that uses key-dependent transformations, in this paper we focus on an implementation of the North Korean cipher Pilsung [Kry18a] as reverse-engineered by Kryptos Logic [Kry18b]. We thus investigate the following question:

Do Pilsung's key-dependent transformations protect it from cache-based attacks?

To tackle this question, we mount the first cache attack on an implementation of Pilsung.

Overview of Pilsung. Pilsung is an AES-like cipher, which uses a different S-Box for each byte at each round. These S-Boxes, as well as the affine permutation in each of the rounds, are generated using information from the secret key. Consequently, in the absence of knowledge about the secret key, an attacker has to find the specific S-Boxes and permutations used. Furthermore, because each S-Box entry is a single byte, the amount of information leaking via cache access is very limited, as a cache access only reveals two bits of information about the index.

Attacking Pilsung. Despite these challenges we show two separate attacks. The first attack completely recovers the first round key by exploiting the lack of cache alignment in the public Pilsung implementation. We extend the works on misaligned tables [SP13; ZW10] and show how the alignment of the tables affects the amount of information leaked.

Noting that the misaligned tables may be an artifact of the reverse engineering of Pilsung rather than a flaw in the original implementation, we also present an attack on Pilsung when the tables are properly aligned with cache lines.

Attack Model. To decouple the analysis of the cipher from the intricacies of cache attacks, we model the cache attack as an oracle. The oracle observes the behavior of the cipher and returns the cache-line index of a specific S-Box access during the encryption process. We then show how to implement the oracle through a Prime+Probe [OST06] cache attack via the Mastik [Yar16] framework.

Attack Overview. Our attack first reverses the pseudo random permutation used in Pilsung. In a nutshell, like AES, Pilsung uses a two-step permutation. It first permutes the bytes of the state, and then mixes the values of groups of bytes. Following the standard AES notation, we use the names *Shift-Rows* and *Mix-Columns*. However, unlike AES, the Shift-Rows step in Pilsung uses a key-dependent permutation of the bytes of the state.

To reverse this permutation, we first brute force the mapping of state bytes to columns. For that we select multiple plaintexts in which four of the bytes are *fixed*, i.e. have the same value in all plaintexts. Fixing bytes fixes the corresponding indices accessed in the first round. Furthermore, if the permutation moves all four fixed bytes to the same column, fixing the bytes fixes the indices accessed in the second round. By counting the number of fixed indices we can identify the four plaintext bytes that map to each column.

After reversing the permutation, we proceed to recover the key. Our attack exploits the limited dependency of the first round S-Boxes on the second round key. Specifically, the S-Box used for the n^{th} byte in the first round solely depends on the corresponding byte of the second round key. Thus, for the attack, we guess the two corresponding bytes in the first and second round keys, and verify the guess by checking the propagation of the plaintext values of the same byte to second round cache accesses. Overall, our attack requires 19472 oracle queries to fully break the cipher.

Overall Attack Performance. To perform the attack, we use the Mastik [Yar16] implementation of the Prime+Probe attack [OST06]. Because the attack only achieves a noisy version of the oracle, we need to encrypt multiple plaintexts for each oracle query. Overall, we achieve complete key recovery by monitoring 3.52×10^7 encryptions.

Summary of Contributions. In this paper we make the following contributions:

- We perform the first cache attack on an implementation of Pilsung, a cipher that uses key-dependent transformations. We show that in Pilsung's case, key-dependent transformations do not provide an inherent protection against side channel attacks (Sections 4 and 5).
- We show how to use a cache attack to reverse engineer secret permutations in AES-like ciphers (Section 4.3).
- We show that the large number of S-Boxes supported in the Pilsung implementation, as reverse-engineered by Kryptos Logic [Kry18b], actually facilitates efficient cache-based attacks (Section 4.4).
- As an independent contribution, we investigate the effects of misaligned tables on cache attacks and provide a formula for calculating the amount of leakage due to misalignment (Section 3).

2 Background

2.1 Cache Attacks

To bridge the speed gap between the fast processor and the slower memory, processor designers introduce small banks of fast memory called *caches*. Caches store recently accessed *cache lines*—blocks of memory of a fixed size, typically 64 bytes—exploiting the temporal and spatial patterns of memory accesses often found in software.

Observing Cache Timing Difference. While caches are functionally transparent to the application software, the speed difference between accessing data in the cache (*cache hit*) and data not in the cache (*cache miss*) can reveal information on the contents of the cache. Moreover, because the contents of the cache depends on prior computation, recovering information on the contents of the cache may disclose secret information on prior computation. Multiple techniques for exploiting this timing difference have been developed, showing the effectiveness of *cache attacks* in recovering keys of symmetric ciphers [AK06; Ber05; BM06; IAES15; IAIES14; MIE17; NRMW12; NS06; NSW06; OST06; PDR11; RM11; RMTF09; SP13; TSS+03; TTMH02; ZW10; ZWZ09], public key ciphers [BBG+17; BH09; GBK11; LYG+15; Per05; RGG+19; YF14; ZJRR12], digital signature schemes [BPSY14; PBY17; PGBY16], zero-knowledge proofs [DDME+18] and hybrid schemes [GVY17]. Cache attacks have also been shown to be effective in recovering sensitive information from non-cryptographic software [BMD+17; GSM15; RTSS09; SKH+19; YFT18].

2.2 Prime+Probe.

Prime+Probe [OST06] is a cache attack technique that exploits the structure of the cache to leak information. Modern caches are typically *set associative*, meaning that the cache is divided into multiple *sets*, such that each set can only store data from a subset of the physical memory. The Prime+Probe attack first *primes* one or more cache sets, filling them with the attacker's data. The attack then allows the victim to execute, before finally *probing* the cache, i.e. measuring the time it takes to access the previously cached data. Slow access indicates that the victim execution replaced some of the previously cached data, causing memory access in the prime stage. Thus, through monitoring cache sets in which the victim execution replaces data, the attacker can learn that the victim accessed data in memory locations that map to these cache sets.

2.3 Prime+Probe on Block Ciphers

Overview of Block Ciphers. Block ciphers are typically implemented as a sequence of *rounds* that operate on an internal *state*. For encryption, the state is initialized with the plaintext. Each round applies a function that mixes key material, i.e. some bits that depend on the key, with the current state to produce the next state. The final state is the ciphertext. The process is reversed for decryption.

Security via Non-Linearity. Because linear functions can be reverted trivially, the round function of ciphers include non-linear components. Many ciphers use *S-Boxes*, which map short sequences of bits to other sequences of bits. S-Boxes can be implemented as lookup tables, such that the n^{th} table entry for S-Box S contains the value $S(n)$. To speed up the round function, some ciphers store precomputed values in *T-Tables*.

Extracting Table Indices. Cache attacks aim to find information on the indices used for specific table accesses. For example, in AES the indices of first round accesses to S-Boxes or T-Tables are the exclusive-or (XOR) of each plaintext byte with the corresponding key byte. Hence, finding the index used at the first round, combined with a known plaintext, allows the attacker to recover the key.

The Prime+Probe attack, described above, can find the cache set that a victim program accesses. As cache lines are typically much larger than the lookup table entries, Prime+Probe typically recovers only partial information on the table index used. Specifically, when the S-Boxes input and output are both 8 bits, and the cache line is 64 bytes, identifying

the cache line accessed for a specific plaintext discloses as little as two bits of information on the corresponding key byte.[1]

Synchronous vs. Non-Synchronous Attacks. Cache attacks tend to have limited spatial and temporal resolution. The literature distinguishes between two attack settings. In a *synchronous* attack, the attacker is able to synchronize the attack with the victim. Typically, the attacker performs the prime stage before the victim encrypts or decrypts a block of data, and the probe step after the encryption or decryption completes. In an *asynchronous* attack, the attacker executes concurrently with the victim, but cannot a-priori control the relative ordering of the operations.

In this work we are interested in the weaknesses of Kryptos Logic's implementation of Pilsung rather than in advancing the techniques for cache attacks. Hence, we focus on the weaker attack model of synchronous attacks.

Handling Partial Information. In a typical attack, the attacker aims to identify a specific access to an S-Box. That is, the attacker knows that the input to the S-Box comprises a known value p combined with an unknown key material k, e.g. by calculating the exclusive or of the two. The attacker then wants to find the cache set accessed in this scenario, and from this find information on the index $p \oplus k$ and recover information on k.

The signal from Prime+Probe tends to be noisy, in part due to the limited temporal resolution, but more so because the victim accesses multiple cache sets during the cipher operation. To distinguish the targeted accesses, the attacker performs multiple Prime+Probe rounds, with different plaintexts. The attacker then averages the access times to each cache set over all of the plaintexts, and over the set of plaintexts that result in the desired state at the input of the S-Box. Because in the latter case the cipher *always* has the same input to the target S-Box, the affected cache set will show a longer average access time when the desired state is achieved.

2.4 Pilsung

2.4.1 Encryption Function

Pilsung is a substitution-permutation network based on AES [Fip]. It manipulates a 128-bit block divided to 16 bytes in 10 rounds, using 11 128-bit round keys. Each 128-bit value is viewed as a byte oriented 4×4 matrix. We denote the 128-bit plaintext by $M = M[i, j]$, and the 128-bit r^{th} round key by $RK^r = RK^r[i, j]$, $i, j \in [0, 3]$, $r \in [0, 10]$. As for AES, the round function consists of the application of four functions:

- **Add-Round-Key.** Each byte of the state is XORed with the corresponding byte of the round key.
- **Sub-Bytes.** Each byte of the state is passed through a non linear layer S-Box. For each round r and each state byte i, j, Pilsung uses a different S-Box in a key-dependent manner, thus leading to a total of 160 different S-Boxes (16 per round, across 10 rounds). We denote by $SB^r_{i,j}$ the S-Box used in round r on state bytes in location i, j.
- **Shift-Rows.** The 16 bytes of the state are rearranged according to a permutation. This permutation is key dependent and is different for each round. We denote by SR^r the permutation used at round r.
- **Mix-Columns.** Each column-wise group of four bytes of the state is multiplied by a 4×4 matrix. This operation is the same as in AES.

[1] We assume here that the tables are aligned with cache lines. See Section 3 for a discussion of misaligned tables.

To ease the explanation, we view the round function as the successive application of Add-Round-Key, Sub-Bytes, Shift-Rows and Mix-Columns. The last round omits the Mix-Columns and performs a final whitening before the output of ciphertext, using the last round key RK^{11}. This is slightly different from the traditional view of AES as consisting of an initial key whitening, followed by rounds of Sub-Bytes, Shift-Rows, Mix-Columns and Add-Round-Key.

The only differences between Pilsung and the standard AES-128 encryption function lie in the key-dependent Sub-Bytes and Shift-Rows operations. For each round r, these are computed using the round key $r+1$ in the following manner:

Setting the Shift-Rows Permutation: Using the 128 bits of RK^{r+1}, a pseudo-random permutation $SR^r = SR^r[i,j]$, $i,j \in [0,3]$ of 16 elements is computed, where $SR^r[i,j] = (i',j')$, i' (resp. j') denoting the new row (resp. column) index of the permutation for the input i,j. The 16 bytes of the state are then permuted according to SR^r. The algorithm for computing SR^r from RK^{r+1} is known [Kry18a], but we do not exploit it for the attack and assume that SR^r is chosen at random.

Setting the S-Boxes: Using the 8-bit value $RK^{r+1}[i,j]$, a pseudo-random permutation P of eight elements is computed. $SB^r_{i,j}(x)$ is then computed by applying the permutation P to the output bits of the standard AES S-Box. The way P (and thus $SB^r_{i,j}(x)$) is computed from $RK^{r+1}[i,j]$ is known and can be found in [Kry18a].

2.4.2 Key Scheduling

The key scheduling is again similar to that of AES, with few differences. First, a 256-bit master key MK is passed through a SHA1-based function, generating 256 bits of key material K. Next, 176 bytes of round keys RK^r are generated from K using the AES key scheduling. However, instead of using $N_k = 4$ (reusing the same notation as in the AES FIPS document [Fip]) as would be expected for a 10 rounds AES-like encryption function, Pilsung uses the key scheduling function with $N_k = 5$ and $N_r = 10$. This means that only the first 160 bits of K are used to generate the round keys.

We make two observations about Pilsung's key scheduling. First, the knowledge of K is enough to completely break the cipher. That is, the 256-bit key MK and the hash function can be ignored. Second, only 160 bits of K are used to generate the round keys. Of these 160 bits, the first 128 are used in the first Add-Round-Key as RK^0, and the next 32 are used in the second Add-Round-Key, as the first 32 bits of RK^1.

2.4.3 Implementation

The application of the S-Boxes is done through memory accesses. That is, the 10×16 S-Boxes are all stored in memory. With each S-Box occupying 256 bytes, the 16 S-Boxes of each round require 4 KB of memory, fitting exactly across a typical 8-way L1 of 32KB while avoiding set collision. This absence of collisions allows us to differentiate between the S-Boxes of the round and facilitates associating cache information with the S-Boxes.

3 Information in Misaligned Tables

We look at the typical first-round attack, where an input byte p is combined with a secret key byte k to form an index $p \oplus k$ to an S-Box.[2] The attacker can observe the cache line of the access to the S-Box, and tries to use such observations to learn information

[2] To save space, we assume fixed sized S-Boxes, as used in AES and Pilsung. Extending to other sizes is straightforward, as long as the sizes of the tables and the table entries are all powers of two.

on k. In the standard scenario, we assume that the S-Box is cache-aligned, i.e. that the address of entry 0 of the S-Box is at the start of a cache line. When cache lines are 2^l bytes long, this implies that entry $p \oplus k$ of the S-Box falls in cache line $\lfloor (p \oplus k)/2^l \rfloor$ of the S-Box. Recovering this reveals the $8 - l$ most significant bits (MSBs) of k. Thus, on Intel machines, where cache lines are $64 = 2^6$ bytes long, we can expect to recover $8 - 6 = 2$ bits of information on each key byte in a first-round attack.

Due to the way that the S-Boxes are allocated in the reverse-engineered version of Pilsung [Kry18b], the S-Boxes are not cache-aligned. Instead, they are offset by some number of bytes.

Several works consider the case that the tables are not cache aligned. Osvik, Shamir and Tromer [OST06] suggest that misaligned tables add one bit of information, but do not elaborate. Zhao and Wang [ZW10] and later Spreitzer and Plos [SP13] further investigate the case of misaligned tables. Both show that in some cases of misaligned tables, the adversary can recover the secret completely, but stop short of analyzing the relationship between the misaligned tables and the number of bits that can be recovered.

We now show that if the S-Box is offset by $0 < \delta' = 2^l - \delta < 2^l$ bytes, for $\delta = (2\alpha+1)2^\beta$, the attacker can recover all but the β least significant bits (LSBs) of k.

3.1 Recovering the Most Significant Bits

We begin by observing that each group of 2^l values of $p \oplus k$ falls within two consecutive cache lines instead of a single cache line. Specifically, δ of them fall in the cache line with the smaller index, and the rest in the other. We now show how we can use this observation to recover the $8 - l$ MSBs of k.

As the S-Box is offset by δ', we have that the table entry $p \oplus k$ is located at cache line $A = \lfloor ((p \oplus k) + \delta')/2^l \rfloor$. Let p be such that $p \oplus k$ that is a multiple of 2^l, i.e. $p \oplus k$ has the l LSBs set to zero. For such a p, it holds that

$$A = \lfloor ((p \oplus k) + \delta')/2^l \rfloor = \lfloor (p \oplus k)/2^l \rfloor.$$

This is because the addition of $\delta' < 2^l$ to $p \oplus k$ can only affect the lowest l bits of $p \oplus k$, and no carry is generated because these bits are zero. Thus, assuming the attacker knows a value of p such that $p \oplus k$ is a multiple of 2^l, the $8 - l$ MSBs of k can be recovered using

$$\lfloor k/2^l \rfloor = \lfloor (p \oplus k)/2^l \rfloor \oplus \lfloor p/2^l \rfloor = \lfloor ((p \oplus k) + \delta')/2^l \rfloor \oplus \lfloor p/2^l \rfloor = A \oplus \lfloor p/2^l \rfloor = A.$$

For an unknown k, the attacker cannot generally obtain a p such that $p \oplus k$ is a multiple of 2^l. However, there exists one such p in the range $0, 1, \ldots, 2^l - 1$. As discussed above, this range maps to two cache lines, and we know that for this value of p, the access will fall in the lower indexed cache line. Hence, observing an access to the lower line reveals the value of A, and solving above equation above reveals the $8 - l$ MSBs of k.

3.2 Recovering Other Bits

We now proceed to show how we use the observation that the values of $p \oplus k$ fall within two cache lines to recover more bits of k. First, as δ values of $p \oplus k$ fall in the cache line with the smaller index, we can obtain a set $P = \{p_0, p_1, \ldots, p_{\delta-1}\}$ such that $(p_i \oplus k) \bmod 2^l < \delta$ for all $0 \le i < \delta$. We now show that given a k' such that $(p_i \oplus k') \bmod 2^l < \delta$ for all $0 \le i < \delta$ we have $\lfloor k/2^\beta \rfloor \bmod 2^{l-\beta} = \lfloor k'/2^\beta \rfloor \bmod 2^{l-\beta}$, that is, k and k' differ only in the β LSBs. Having established this, the adversary can test all 2^l possible values of k' to find at least one such that $(p_i \oplus k') \bmod 2^l < \delta$ for all $0 \le i < \delta$, and from this k' get all but the β LSBs of k.

We first note that because $\delta = (2\alpha + 1)2^\beta$, for all $0 \le x < 2^l$ we have

$$x < \delta \text{ if and only if } \lfloor x/2^\beta \rfloor < \delta/2^\beta = 2\alpha + 1. \tag{1}$$

There exists $p_i \in P$ such that $(p_i \oplus k) \bmod 2^l = \delta - 1$. We therefore obtain that $(\delta - 1) \oplus k = p_i \in P$. Thus, it holds that

$$(\delta - 1) \oplus (k \oplus k') = p_i \oplus k' < \delta \tag{2}$$

where the last transition follows from our assumption regarding k' and the associativity of exclusive-or. Because $\delta = (2\alpha + 1)2^\beta$, we have that $\delta - 1 = \delta \oplus (2^{\beta+1} - 1)$. Combining this with Equation 2, we obtain that

$$(\delta - 1) \oplus (k \oplus k') = \delta \oplus (2^{\beta+1} - 1) \oplus (k \oplus k') < \delta.$$

Next, applying Equation 1 we obtain that $\lfloor (\delta \oplus (2^{\beta+1} - 1) \oplus (k \oplus k'))/2^\beta \rfloor < (2\alpha + 1)$, or equivalently

$$(2\alpha + 1) \oplus 1 \oplus \lfloor (k \oplus k')/2^\beta \rfloor < (2\alpha + 1). \tag{3}$$

Conversely, as for all $p_i \in P$ it holds that $(p_i \oplus k) \bmod 2^l < \delta$ we have that $\delta \oplus k \notin P$. Hence, $\delta \oplus (k \oplus k') \geq \delta$, and by Equation 1 $\lfloor (\delta \oplus (k \oplus k'))/2^\beta \rfloor \geq \delta/2^\beta$, or

$$(2\alpha + 1) \oplus \lfloor (k \oplus k')/2^\beta \rfloor \geq (2\alpha + 1). \tag{4}$$

However, Equations (3) and (4) can both be true only if $\lfloor (k \oplus k')/2^\beta \rfloor = 0$. Thus all key hypotheses that agree with the set P agree on all but the β least significant bits.

Specifically, due to the layout of the Pilsung context, the S-Boxes are all in odd offsets, i.e. $\beta = 0$. Hence, a first round attack can recover all of the key bits.

4 A Theoretical Attack on Pilsung

We now investigate the more complicated case where the tables are aligned in memory. For the theoretical attack, we assume that the attacker has access to an oracle $O(M, K, r, i, j)$ that, given the plaintext M and the key K, returns the two most significant bits (MSBs) of the S-Box input at round r for the state byte i, j. We show how the attacker can use the oracle to recover the full 160 bits of secret key. We start by introducing new notations. We then show how much information can be obtained by attacking the first round. We then show how to use the oracle to reverse the key-dependent Shift-Rows permutation of the first round. Finally, we target the second round, recovering the secret key.

4.1 Notations

To simplify the explanations of the different attack steps, we introduce additional notations for the Pilsung's state. We use $A^r = A^r[i, j]$, $i, j \in [0, 3]$ to denote the state matrix before the Add-Round-Key (AK) operation of round r. Similarly, B^r, C^r and D^r respectively denote the matrix state at round r before the Sub-Bytes (SB), Shift-Rows (SR) and Mix-Columns (MC) operations. The state is initialized with $A^0 = M$. Figure 1 illustrates this representation.

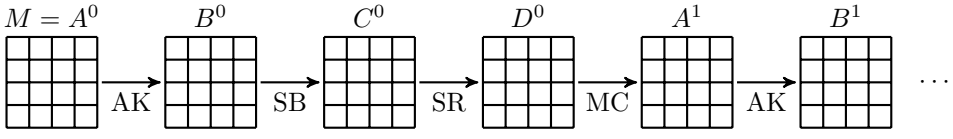

Figure 1: Pilsung states after the different operations and rounds. AK, SB, SR, and MC stand for Add-Round-Key, Sub-Bytes, Shift-Rows, and Mix-Columns, respectively.

Using these notations, we can write the output of the oracle as $O(M, K, r, i, j) = (B^r[i, j] >> 6)$, where $x >> r$ is the value x after a right-shift of r bits, i.e. $x >> r = \lfloor x/2^r \rfloor$.

4.2 First Round Attack

The first round attack essentially follows the same procedure as the standard Prime+Probe attack described in [OST06]. That is, for a given target byte i, j, the attacker simply needs to query the oracle once in the first round. Such query gives the attacker the two MSBs of $B^0[i, j]$, the input of the S-Box i, j at round 0. As this value only depends on one byte of the plaintext and of the first round key, the attacker straightforwardly obtains the two most significant bits of $RK^0[i, j]$. More formally, Algorithm 1 describes the procedure to attack the first round from the oracle.

Algorithm 1 First round attack.

Input: Plaintext M, indices i, j, oracle O
Output: r, the 2 MSBs of $RK^0[i, j]$
1: $v \leftarrow O(M, K, 0, i, j)$
2: $r \leftarrow (m_{i,j} >> 6) \oplus v$
3: **return** r

repeating Algorithm 1 for each of the 16 bytes' indices, the attacker obtains the two MSBs of each of the first round key bytes. As a result, the entropy is reduced from 160 bits to 128. Note that no further information can be obtained by analyzing the first round.

4.3 Reversing the Shift-Rows Permutation

Attacking the first round does not give enough information to break the cipher. To obtain information on the remaining key bits, the attacker has to go deeper in the cipher and target the second round. However, as the Shift-Rows operation is key dependent, the attacker cannot directly predict which bytes of the plaintext are going to affect which bytes of the second round S-Boxes. Consequently, the attacker first needs to recover the key-dependent permutation used during the first Shift-Rows.

We use a two step procedure for recovering the key-dependent permutation. We first find which set of four bytes is mapped to the which column after the Shift-Rows operation, without recovering the rows thy mapped to in the column. In the second step we find the ordering of these four bytes within this column. For the attack we do not make any assumption about how the permutation is generated. Thus, the procedure is also applicable to random permutations.

4.3.1 Column Mapping

As previously mentioned, the first step consists of finding which group of four bytes before Shift-Rows will be mapped to the same column after the application of the permutation. More formally, for each $j \in [0, 3]$ we aim to find four indices (i_k, j_k), $k \in [0, 3]$ such that $SR^0[i_k, j_k] = (i'_k, j)$. See Figure 2 for a graphical representation of the goal of this step. Each color represents a group of four bytes that are mapped to the same column after the unknown SR. We want to find the color-wise mapping. However, we do not try to recover the ordering within a given color. More formally, we do not (yet) find the values of i'_k.

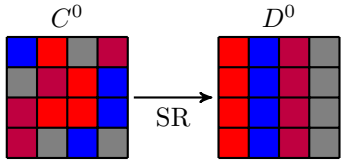

Figure 2: Illustration of the column mapping step.

To perform this column mapping, we query the oracle at the second round, revealing information on the MSBs of each byte of B^1, the inputs of the second round S-Boxes. We start with the following observation: if (a) at least one byte of each column in D^0 is random, then the Mix-Columns operation will propagate this randomness among the bytes in A^1, and thus B^1. However, if (b) a given column index j of D^0 is fixed, then the corresponding column in B^1 will remain fixed as well. These two cases are illustrated in Figure 3. The top (resp. bottom) part of the figure shows the case a (resp. b). Colored bytes correspond to fixed values, while white bytes correspond to randomly varying bytes.

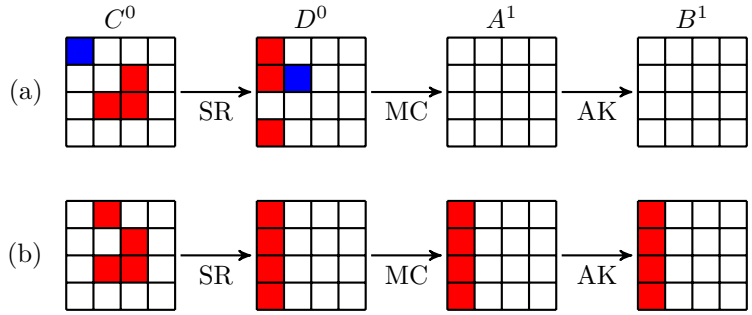

Figure 3: Illustration of the two propagations cases with four fixed bytes.

The knowledge of which situation, (a) or (b), happened can be used to reverse the column mapping of the permutation. To do this, we first choose a set of set of four bytes of the plaintext M. We then fix the values of these four bytes and set the 12 others as random for several cipher executions. If case (b) occurs, we know that the indices of the four fixed bytes are mapped to the same column j after the Shift-Rows operation. However, if case (a) occurs, this means that at least one of the four fixed bytes' indices was not mapped within the same column as the other three after the Shift-Rows operation.

To find which case occurs, we query the oracle in the second round with $n > 1$ different plaintexts M_i, such that four randomly chosen bytes are identical for all i, and with the remaining 12 bytes set as random. For each plaintext, the attacker queries the oracle for the entire state matrix. If case (b) occurs, at least one column will have the same four oracle answers for all plaintexts. The oracle queries of the other column will result in different answers with high probability. More precisely, as the oracle answer is two bits, the probability that another column shows the same answer over all n plaintexts for all the four bytes is $2^{-8(n-1)}$. Thus, if case (a) occurs, it is likely that the oracle queries will show differences for all column. Increasing the number of plaintexts n reduces the likelihood of false positives.

Putting Things Together: The overall strategy to perform the column mapping consists of randomly fixing four bytes and then querying the oracle to detect if case (a) or (b) happened. If case (b) is detected, then the adversary knows which columns the corresponding indices of the fixed bytes will map to after Shift-Rows. If case (a) is detected, the adversary repeats the operation by fixing different byte indices until case (b) is detected. The probability to randomly find a group of four bytes mapping to a given column is $\binom{16}{4}^{-1} = \frac{1}{1820}$. As there are four columns, the probability to randomly find a column mapping is thus $\frac{4}{1820} = \frac{1}{455}$. Overall, as we are querying the full state for n plaintexts, the attacker will have (roughly) 50% chances to find a column mapping in $n \times 16 \times 223 = 3568n$ queries. As there are 4 columns, and as each column might produce a false positive with probability $2^{-8(n-1)}$ when using n plaintexts, the overall probability of not getting a false positive is $p = (1 - 2^{-8(n-1)})^{4\times 223}$. As a result, the probability that at least one column will exhibit a false positive among all the executions is $1 - p$. While for two plaintexts

Algorithm 2 Column mapping procedure.

Input: Oracle O, n the number of plaintexts
Output: Indices (i_k, j_k) and fixed J s.t. $SR^0[i_k, j_k] = (i'_k, J)$

loop
  $(i_k, j_k) \leftarrow$ *random* for $0 \leq k < 4$
  $M \leftarrow n$ 4×4 matrices $M_0, \cdots, M_{n-1}$ such that

$$M_l(i,j) = \begin{cases} 0 & \text{if } (i,j) = (i_k, j_k) \text{ for some } k \\ \textit{random} & \text{otherwise} \end{cases}$$

  $R \leftarrow n$ 4×4 empty matrices R_l, $l \in [0, n-1]$
  for all $0 \leq i, j < 4, 0 \leq l < n$ **do**
    $R_l[i,j] \leftarrow O(M_l, K, 1, i, j)$
  for J=0 to 3 **do**
    if $R_0[i,J] = R_l[i,J]$ for all $0 \leq i < 4, 1 \leq l < n$ **then**
      return $(i_k, j_k), J$

only the probability to get a false positive among all the executions is as high as 0.97, it quickly drops to 2×10^{-7} for five plaintexts. Finding the next column mapping is simpler since we already found the previous ones: the probabilities increase to $\binom{12}{4}^{-1} = \frac{1}{495}$ for the second one, $\binom{8}{4}^{-1} = \frac{1}{70}$ for the third, and finally 1 for the last one.

More formally, Algorithm 2 describes the procedure to find the first column mapping. $a \leftarrow$ *random* sets a to a random value between 0 and 255.

4.3.2 Column Ordering

From the previous method, we now know how to find which group of four bytes is mapped to a single column after the key-dependent Shift-Rows operation. However, we still do not know the byte ordering within this column. Without loss of generality, we will now assume, for simplicity, that the Shift-Rows operation maps the first column to itself. That is, we know that $SR^0[i_k, 0] = (i'_k, 0)$, $k \in [0, 3]$. We now aim to finding the corresponding mappings $i_k \rightarrow i'_k$. For more clarity, Figure 4 represents the goal of this step. That is, we aim at finding which indices map to which before and after the Shift-Rows operation for the first column. Naturally, we will repeat this process for the remaining three columns.

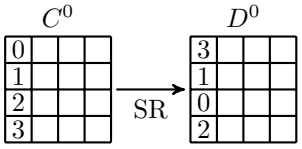

Figure 4: Illustration of the column ordering step.

For that purpose, we again look at the memory accesses from the second round S-Boxes. We now show that when fixing three out of these four bytes to some constant, the four corresponding cache accesses in the second round allow us to reveal the exact position of the fourth varying byte after the Shift-Rows operation.

Still assuming that Shift-Rows maps the first column to itself, Equation 5 shows the values of $B^1[i, 0]$, $i \in [0, 3]$, which are the four inputs used to access the S-Boxes during the second round. The operator $\times$ denotes the multiplication over $\mathbb{F}_{256}$.

$$\begin{bmatrix} B^1[0,0] \\ B^1[1,0] \\ B^1[2,0] \\ B^1[3,0] \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} D^0[0,0] \\ D^0[1,0] \\ D^0[2,0] \\ D^0[3,0] \end{bmatrix} \oplus \begin{bmatrix} RK^1[0,0] \\ RK^1[1,0] \\ RK^1[2,0] \\ RK^1[3,0] \end{bmatrix} \tag{5}$$

And we have

$$\begin{bmatrix} D^0[0,0] \\ D^0[1,0] \\ D^0[2,0] \\ D^0[3,0] \end{bmatrix} = \begin{bmatrix} C^0[SR^0[0,0]] \\ C^0[SR^0[1,0]] \\ C^0[SR^0[2,0]] \\ C^0[SR^0[3,0]] \end{bmatrix} \tag{6}$$

If we fix $C^0[SR^0[1,0]]$, $C^0[SR^0[2,0]]$ and $C^0[SR^0[3,0]]$, we can then rewrite $B^1[i,0]$ as a function of $C^0[SR^0[0,0]]$:

$$\begin{aligned} B^1[0,0] &= 2 \times C^0[SR^0[0,0]] \oplus cst_0 \\ B^1[1,0] &= C^0[SR^0[0,0]] \oplus cst_1 \\ B^1[2,0] &= C^0[SR^0[0,0]] \oplus cst_2 \\ B^1[3,0] &= 3 \times C^0[SR^0[0,0]] \oplus cst_3 \end{aligned} \tag{7}$$

where cst_i denote unknown constants that depend on $RK^0[i,0]$ and $SB^0_{i,0}$, $i \in [1,3]$. We can observe that up to some constant XOR both $B^1[1,0]$ and $B^1[2,0]$ directly depend on $C^0[SR^0[0,0]]$, while $B^1[0,0]$ (resp. $B^1[3,0]$) depends on $2 \times C^0[SR^0[0,0]]$ (resp. $3 \times C^0[SR^0[0,0]]$). Such dependency can be observed when looking at the cache access patterns. That is, the cache accesses from $SB^1_{1,0}$ and $SB^1_{2,0}$ will be exactly the same up to a permutation given by the two MSBs of $cst_1 \oplus cst_2$. More formally, we have $O(M,K,1,1,0) = O(M,K,1,2,0) \oplus cst_1 \oplus cst_2$. However, due to the non-linearity of the multiplication by 2 and 3, the cache accesses from $SB^0_{0,0}$ and $SB^0_{3,0}$ will be different from the ones from $SB^0_{1,0}$ and $SB^0_{2,0}$.

Table 1: Attack contexts.

	$2x \oplus \texttt{0x}b8$	$x \oplus \texttt{0x}85$	$x \oplus \texttt{0x}32$	$3x \oplus \texttt{0x}df$
$x = \texttt{0x}00$	0b10	0b10	0b00	0b00
$x = \texttt{0x}90$	0b10	0b00	0b10	0b10
$x = \texttt{0x}60$	0b01	0b11	0b01	0b10
$x = \texttt{0x}f8$	0b01	0b01	0b11	0b00

Table 1 illustrates this phenomenon, where 0x and 0b denote hexadecimal and binary notations, respectively. The variable x represents $C^0[SR^0[0,0]]$. We arbitrarily set the constants $cst_0 = \texttt{0x}b8$, $cst_1 = \texttt{0x}85$, $cst_2 = \texttt{0x}32$ and $cst_3 = \texttt{0x}df$. As a result, each column corresponds to the value computed in $B^1[i,0]$. For four different arbitrarily chosen values of x, the table shows the two MSBs of the given operation, which corresponds to the oracle response. That is, the column i corresponds to $O(M,K,1,i,0)$. As we can see, we have $O(M,K,1,1,0) = O(M,K,1,2,0) \oplus \texttt{0b10}$, and $\texttt{0b10} = (\texttt{0x}85 \oplus \texttt{0x}32)\texttt{>>}6$, Interestingly, we can detect other specific patterns if the only varying byte is at another position than $D^0[0,0]$. For example, if $C^0[SR^0[0,0]]$ is the only varying value, we would observe a similar cache access pattern for both $SB^1_{2,0}$ and $SB^1_{3,0}$.

This means that when fixing three out of the four bytes within this column, the cache access patterns in the second round reveal information about which bytes are fixed after the Shift-Rows operation, thus revealing the mapping of the one byte that is left as varying.

Algorithm 3 Column ordering procedure.

Input: Oracle O, byte index x to map, n number of plaintexts
Output: Index y such that $C^0[x,0]$ maps to $D^0[y,0]$.

$L_0, L_1, L_2, L_3 \leftarrow Array(n)$
for $i = 0$ to 3 **do**
  if $i \neq x$ **then**
    $M[i,0] \leftarrow 0$
for $i = 0$ to $n-1$ **do**
  $M[x,0] \leftarrow random$
  for $j = 0$ to 3 **do**
    $L_j[i] \leftarrow O(M, K, 1, j, 0)$
return y such that $L_{y+1 \bmod 4} = L_{y+2 \bmod 4} \oplus cst$

If we reuse the example of Figure 4, the attacker can decide to fix the bytes of C^0 with indices 1, 2 and 3 and leave the byte of index 0 as varying for several plaintexts. After the Shift-Rows operation, only the byte $D^0[2,0]$ in the first column will be varying, and the other ones being fixed. Obviously, this information is not yet known by the attacker. They would then query the oracle for accesses to all four S-Boxes $SB^1_{0,0}$, $SB^1_{1,0}$, $SB^1_{2,0}$, $SB^1_{3,0}$. With the same reasoning as the equations above, they will observe that $SB^1_{0,0}$ and $SB^1_{3,0}$ will trigger the same cache access pattern up to some constant, while the other two S-Boxes will show completely different patterns. Such a pattern is the footprint that only $D^0[2,0]$ is varying. The adversary would then know that $C^0[0,0]$ maps to $D^0[2,0]$. Note that several plaintexts are necessary to identify the different patterns of $SB^1_{1,0}$ and $SB^1_{2,0}$. That is, given n plaintexts each S-Box has a probability of 2^{-2n} to incorrectly exhibit the same pattern has the $SB^1_{0,0}$ and $SB^1_{3,0}$. This probability can be arbitrarily reduced by increasing the number of plaintexts. Repeating this procedure with the other three bytes of C^0 will completely reveal the ordering of the first column.

More formally, the procedure to recover the column ordering is given by Algorithm 3. Note that the described algorithm is written assuming (without loss of generality) that the first column of C^0 maps to the first column of D^0. It can be trivially adapted to the needed column mapping. $Array(n)$ denotes a function that returns an array of size n. $a \leftarrow random$ sets a to a random value between 0 and 255.

4.4 Second Round Attack

The previous section shows how to fully recover the unknown permutation used during the first Shift-Rows operation. From that knowledge, we can now exploit the memory accesses of the second round S-Boxes to recover information about the first and second round keys. More specifically, we show how we can use the oracle in the second round to recover the key bytes of the first and second rounds in a divide and conquer manner. For simplicity, and without loss of generality, we assume that the first Shift-Rows operation is the identity mapping, and thus that $C^0 = D^0$.

We start by showing how to recover both $RK^0[0,0]$ and $RK^1[0,0]$. For that purpose, Equation 8 shows the input value $B^1[0,0]$ of the first S-Box $SB^1_{0,0}$ in the second round.

$$\begin{aligned} B^1[0,0] = {} & 2 \times SB^0_{0,0}(M[0,0] \oplus RK^0[0,0]) \\ & \oplus 3 \times SB^0_{1,0}(M[1,0] \oplus RK^0[1,0]) \\ & \oplus SB^0_{2,0}(M[2,0] \oplus RK^0[2,0]) \\ & \oplus SB^0_{3,0}(M[3,0] \oplus RK^0[3,0]) \oplus RK^1[0,0] \end{aligned} \tag{8}$$

Algorithm 4 Second round attack.

Input: Oracle O, target byte index i, j , n number of plaintexts
Output: Key candidates couples for $\left(RK^0[i,j], RK^1[i,j]\right)$.

```
1: CandidateList ← EmptyList
2: L ← Array(n)
3: for all k0 ∈ K0, k1 ∈ K1 do
4:    L_{k0,k1} ← Array(n)
5: for i = 0 to n − 1 do
6:    M ← RandomMsg
7:    L[i] ← O(M, K, 1, i, j)
8:    for all k0 ∈ K0, k1 ∈ K1 do
9:       L_{k0,k1}[i] = B'_{M[i,j],k0,k1} >>6
10: for all k0 ∈ K0, k1 ∈ K1 do
11:    if L_{k0,k1} = L ⊕ cst then
12:       CandidateList ← ADD(k0, k1)
13: return CandidateList
```

If we fix $M[1,0]$, $M[2,0]$ and $M[3,0]$, we can rewrite the equation as follow:

$$B^1[0,0] = 2SB^0_{0,0}(M[0,0] \oplus RK^0[0,0]) \oplus RK^1[0,0] \oplus cst \tag{9}$$

where cst denotes some unknown constant that depends on $RK^0[i,0]$ and $SB^0_{i,0}$, $i \in [1,3]$. However, as this value is constant, the cache access of $SB^1_{0,0}$ that depends on $B^0[0,0]$ will show the same pattern as the value $B' = B^0[0,0] \oplus cst$:

$$B' = 2SB^0_{0,0}(M[0,0] \oplus RK^0[0,0]) \oplus RK^1[0,0] \tag{10}$$

We note that B' only depends on $M[0,0]$, $RK^0[0,0]$ and $SB^0_{0,0}$, which in turn depends on $RK^1[0,0]$. By trying all the possible values of $RK^0[0,0]$ and $RK^1[0,0]$, we can thus predict the corresponding value of B'. For the actual subkey guesses, we know that the predicted B' and the observed cache accesses of $SB^1_{0,0}$ will exhibit the same pattern. However, for wrong subkey guesses, the non-linearity of $SB^0_{0,0}$ will show a difference between the predicted B' and the observed cache accesses. Assuming that n plaintexts are used, each pair of key candidate for $RK^0[0,0]$ and $RK^1[0,0]$ (among the 2^{16} ones, or 2^{14} if the first round attack has been performed) has 2^{-2n} chance to incorrectly exhibit the same cache pattern as B', thus being incorrectly classified as a correct key. The number of false positives can be arbitrarily reduced by increasing the number of plaintexts n. Naturally, the same procedure can be repeated to recover the remaining pairs $(RK^0[i,j], RK^1[i,j])$.

The procedure to attack the second round is given by Algorithm 4. $B'_{x,y,z} = 2 \times SB(x \oplus y)$ denotes a guess on the value B' when the message is equal to x, the first round key byte is equal to y, and the second round key byte affecting the S-Box SB is equal to z. $\mathcal{K}_0$ and $\mathcal{K}_1$ respectively denote the space of the key bytes i, j of the first and second round. Note that the algorithm has been written assuming that the first Shift-Rows is the identity mapping. It can, however, trivially be adapted to any known permutation.

4.5 Complexity Summary

In this section, we briefly summarize the complexity requirement in terms of oracle queries of each of the different steps. The variable n represents the (tunable) number of plaintexts used for a given attack step.

- **First round attack:** In order to recover the 2 MSBs of a given byte $RK^0[i,j]$, the first round attack simply queries the oracle once in the first round. Repeating this procedure for each byte concludes the attack, which is completed in 16 oracle queries.
- **Column mapping:** The first step of the recovery of SR^0 requires randomly finding sets of 4 bytes that are mapped to the same column after the permutation. It additionally requires repeating the procedure for n plaintexts in order to reduce the probability of false positives. The first column mapping will need $(16 \times 223)n$ oracle queries. Similarly, the second and the third ones will require $(12 \times 83)n$ and $(8 \times 18)n$ queries. We assume $n = 4$ as it already produces a very low probability. Overall, this step requires $(16 \times 223 + 12 \times 83 + 8 \times 18)n = 4708n = 18832$ oracle queries.
- **Column ordering:** The second step of the recovery of SR^0 requires fixing all but one of the bytes of a column, and then querying the oracle for the entire column. This procedure needs to be repeated for n plaintexts in order to avoid false positives. After the first byte ordering is found, the second one only needs to query the oracle for the 3 remaining bytes of the column. As a result, reversing one column requires $(4 + 3 + 2)n$ queries. We again assume $n = 4$ to keep the probability of false positive small. As there are four columns to recover, the overall complexity of this step is $4(4+3+2) = 36n = 144$ oracle queries.
- **Second round attack:** Finally, the second round attack recovers a pair of bytes for both RK^0 and RK^1 by querying the oracle for a fixed byte index in the second round. As it will further be shown in Section 5, $n = 30$ is sufficient to recover the correct pair (note than a lower value could also be used, potentially at the cost of some enumeration). Finally, as this procedure has to be repeated for each byte, the complexity is $16 \times 30 = 480$ oracle queries.

Overall, the number of oracle queries to fully break the cipher is $16+18832+144+480 = 19472$ oracle queries.

5 Experimental Results

We empirically confirm our analysis in Section 4 via a cache attack. As our attack is aimed at highlighting the weaknesses of Pilsung's implementation, we assume the strong attack model where the attacker has sufficient control over the cipher to perform a synchronous attack. To get cache use information, we use the Prime+Probe attack from the Mastik toolkit [Yar16]. All experiments were performed on a MacBook Air (13-inch, Mid 2013 model) with Intel Core i5-4250U CPU processor at 1.30GHz, and 4GB of 1600MHz LPDDR3 memory. The attack targets the L1 data cache of the machine, which, like all modern Intel processors, is a 32 KiB, 8-way set-associative, with 64 sets and with cache lines of 64 bytes.

5.1 S-Boxes Memory Layout

The Pilsung implementation [Kry18b] initializes an array of S-Boxes when setting the key. Although the cipher only uses 10 rounds, the array has room for 30. For each round, the array consists of a 4×4 matrix of S-Boxes, where each S-Box is a 256 byte array.

Due to the way that the implementation allocates the context for the cipher, the S-Boxes array is not aligned in the cache, allowing a trivial key-recovery attack (see Section 3). We correct this, ensuring that each S-Box starts at a cache line boundary. With cache lines of 64 bytes, each S-Box occupies four cache lines, and each round occupies $16 \times 4 = 64$ cache lines. Because the S-Boxes are consecutive in memory, and due to the linear mapping of memory addresses to cache sets, S-Boxes for the same byte of different rounds overlap in the cache.

5.2 First Round Attack

For the first round attack, we generate 10^6 random plaintexts, and perform a synchronous Prime+Probe attack while encrypting each. We calculate the average probe time for each cache set to get the *baseline* cache activity level. To target a key byte, we *split* the results based on the value of the corresponding plaintext byte, and average the probe times for each of the targeted byte's values. Following Osvik, Shamir and Tromer [OST06], we *normalize* the results by subtracting the baseline probe time from the per-value average.

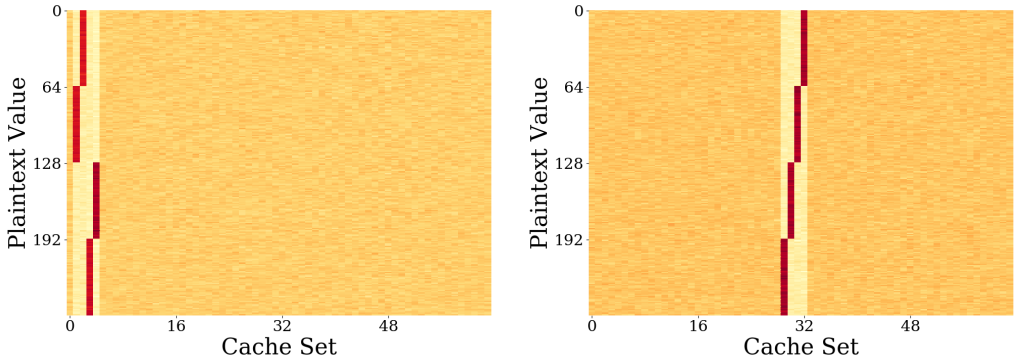

Figure 5: Results of the synchronous Prime+Probe attack on the first round, targeting the byte $[0,0]$ (left) and $[1,3]$ (right). Bytes $[0,0]$ (left) and $[1,3]$ (right) of the first round attack. The two MSBs of $RK^0[0,0]$ (resp. $RK^0[1,3]$) are 01 (resp. 11)

Figure 5 shows the probe results when targeting bytes $[0,0]$ and $[1,3]$ of the key. The horizontal axis is the cache set number, the vertical is the value of the targeted byte, and the shade indicates the normalized probe time of the cache set for the byte value. Darker shades are slower probes.

As we can see, the activity in each of the cache sets is mostly uniform, except for four cache sets. In these cache sets, which cover the S-Box of the targeted byte, we see that one cache set shows more activity (i.e. longer probes) than the others. The cache set that shows more activity provides us with the first round oracle for the targeted bytes. For example, if the first cache set is active, we have $O(M,K,0,i,j)=0$.

Observing the results for the byte $[0,0]$, we see that when the value of the plaintext is between 0 and 63 the third cache set in the S-Box is active. This implies that for $0 \leq M \leq 63$, we have $O(M,K,0,0,0) = 1$, indicating that the top two bits of the corresponding key bytes are 01. Similarly, looking at the right half of the figure, we can determine that the top two bits of the byte $[1,3]$ of the key are 11.

Note that the requirement of 10^6 plaintexts to generate Figure 5 is not needed to complete the attack. Indeed, instead of monitoring the cache access for all 256 values that the target byte can take, one could focus on a single line of the figure to recover the two MSBs of the key, with only $\simeq \frac{10^6}{256}$ plaintexts. Repeating this procedure for all 16 bytes allows the recovery of the two MSBs of $RK^0[i,j]$, $i,j \in [0,3]$, which conclude the success of the first round attack.

5.3 Reversing the Shift-Rows Permutation

As Section 4.3 describes, reversing the Shift-Rows permutation proceeds in two steps. We first brute force the mapping of state bytes to columns and then determine the new order of the bytes in the colums. Figure 6 shows the actual permutation SR^0 that is used in

our experiments. Obviously, that information is unknown to the adversary, who aims at recovering it.

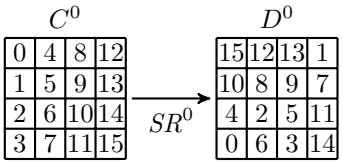

Figure 6: Permutation SR^0 used in the practical experiments.

5.3.1 Column mapping

For the first step, we rely on the observation that if the attacker fixes the four bytes that map to the same column, the state values in the column are fixed in the second round, and consequently, the same cache sets will be accessed in the second round S-Boxes of the column.

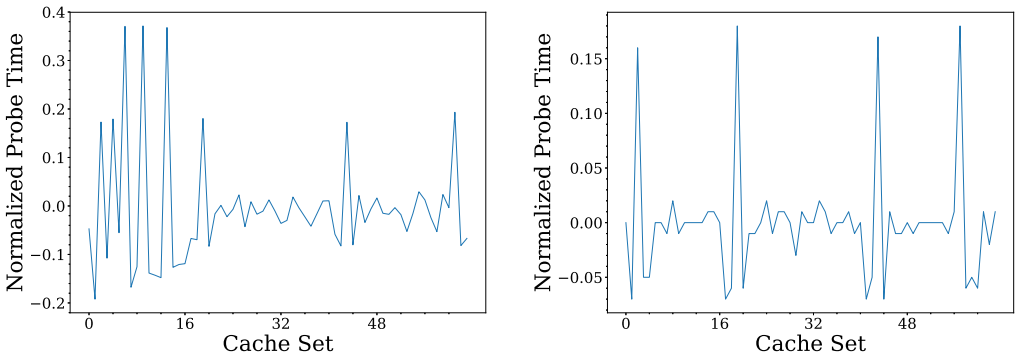

Figure 7: First step in reversing the Shift-Rows. Normalized probe results with four fixed plaintext bytes. Eight cache sets show increased activity when the four bytes map to the same column (left), compared with four when the bytes do not map to the same column (right).

To perform the attack we iterate over all of the $\binom{16}{4}$ combinations of four state bytes. For each such combination we select 10^4 random plaintexts, while fixing the values of the four state bytes we test, i.e. ensuring that the values of each of the four bytes do not change between the selected plaintexts. Note that while this high number of plaintexts originally aims at cancelling the measurement noise, it also directly prevents false positives from happening as shown in Section 4.3.1. We perform Prime+Probe attacks with the selected plaintexts and average the probe times for each of the cache sets. Figure 7 shows the normalized probe times in two cases. The left part of the figure shows the case that all four bytes map to the same column. More precisely, plaintext bytes of indices 15, 10, 4 and 0 are fixed. The right part shows the case where three bytes are in the same column and the fourth in another, by fixing the bytes of indices 14, 10, 4 and 0.

When fixing a byte in the plaintext, we expect the first round to always access the same cache set of the corresponding S-Box. Thus, when fixing four bytes, we expect to see four cache sets that have higher than average activity. Indeed, both parts of the figure show some picks that indicate increased activity. In the right part of the figure we only see the four picks that correspond to the bytes of indices 14, 10, 4 and 0. This corresponds to the case where the *if* condition at line 18 of Algorithm 2 is false, indicating that no

column is fixed after SR^0. Conversely, in the left part, we see increased activity in 8 cache sets. Four of these correspond to the first round S-Box accesses (with indices 15, 10, 4 and 0) and the other four correspond to the accesses of the second round accesses. This corresponds to the *if* condition being true, thus detecting a fixed column after SR^0. As a result, counting the number of cache sets with increased activity identifies the case that all four selected bytes map to the same column. Moreover, the corresponding sets give the position of the fixed column, which is the first one in our case. As a result, we successfully recovered the column mapping from the bytes 15, 10, 4 and 0 to the first column after SR^0. Repeating this operation for the other column finalizes the column mapping procedure.

5.3.2 Column ordering

After determining the state bytes that map to a column, we need to find the order of the bytes within the column. As noted in Section 4.3, we rely on the construction of the Mix-Columns. Specifically, we aim at recovering the ordering within the first column. For that matter, we fix the three plaintext bytes of indices 15, 10, and 4 and leave byte 0 as varying. To identify the new position of this plaintext byte, we perform 10^6 Prime+Probe attacks and then split the results based on the targeted byte. Figure 8 shows the results, with full results on the left side, and highlighted details on the right.

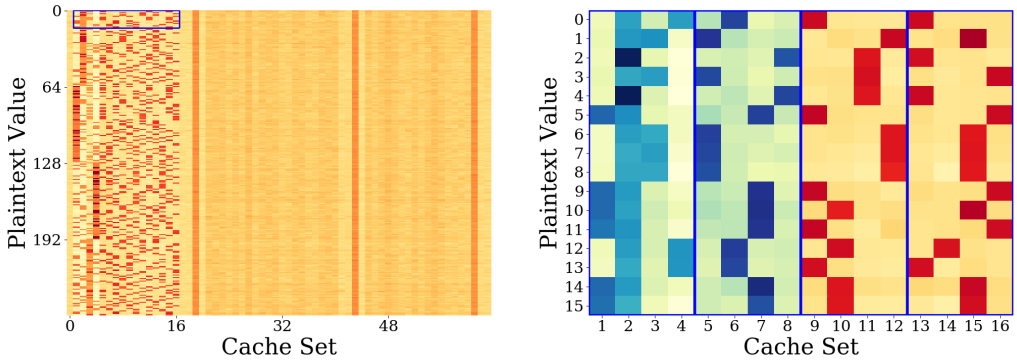

Figure 8: Second step in reversing the Shift-Rows permutation. Full information (left) and marked detail (right). The first and second S-Boxes in the right figure (marked in blue) show the same pattern up to exclusive or with 2.

When the plaintext byte changes, the values of all of the four bytes in the column change, and with them the cache set accessed in the second round. However, as noted in Section 4.3, due to the construction of the Mix-Columns matrix, the effect of the change on two of the bytes will be identical up to exclusive or with a constant, whereas the other two bytes will show different changes.

Observing the cache sets of the first and second S-Boxes highlighted in blue in the right side of Figure 8, we can see that whenever the first cache set is active in the first S-Box, the third cache set is active in the second and vice versa. The same holds for the second and fourth cache sets. That is, the oracle for the first S-Box is the same as for the second S-Box up to exclusive or with 2. This property holds for all plaintext values of the byte, indicating that in Equation 5 the value we vary is multiplied by 1 for the first and second outputs of the Mix-Columns step. Note that we can notice the presence of two active cache sets for the first S-Box. This is due to the fact that the plaintext byte 0 triggers the cache set associated to the first S-Box for both the first and second round of the cipher. However, the corresponding set can be ignored as it is known thanks to the first round attack. Thus, we can conclude that byte 0 is in the last row of the first column

after the application of SR^0, as shown in Algorithm 3. Repeating the process for each of the plaintext bytes allows us to completely recover the Shift-Rows permutation.

5.4 Second Round Attack

Recall (Section 4.4) that for the second round attack, we only need an oracle for the second round access to one of the S-Boxes of the column the targeted byte maps to. We note that Figure 8 already provides the information we require, and is enough to validate the attack. Yet, for completeness, we adapt the formula of Section 4.4 to the actual permutation SR^0 used in our practical experiment. That is, we show in Equation 11 the adaptation of Equation 8, allowing us to recover both $RK^0[0,0]$ and $RK^1[3,0]$,

$$\begin{aligned} B^1[0,0] = 2 \times SB^0_{0,0}(M[3,3] \oplus RK^0[3,3]) \\ \oplus 3 \times SB^0_{1,0}(M[2,2] \oplus RK^0[2,2]) \\ \oplus SB^0_{2,0}(M[0,1] \oplus RK^0[0,1]) \\ \oplus SB^0_{3,0}(M[0,0] \oplus RK^0[0,0]) \oplus RK^1[0,0] \end{aligned} \tag{11}$$

As for the column ordering procedure, we fix the plaintext bytes $M[3,3]$, $M[2,2]$ and $M[0,1]$ and rewrite the equation as follows:

$$B^1[0,0] = SB^0_{3,0}(M[0,0] \oplus RK^0[0,0]) \oplus cst \tag{12}$$

Where cst denotes some unknown constant that depends on $RK^0[i,0]$, $i \in [1,3]$, $SB^0_{i,0}$, $i \in [0,2]$ and $RK^1[0,0]$. Again, as this value is constant, the observed cache access of $SB^1_{0,0}$ will have the same pattern as the value $B' = B^1[0,0] \oplus cst$:

$$B' = SB^0_{3,0}(M[0,0] \oplus RK^0[0,0]) \tag{13}$$

As explained in Equation 11, this relation can be used to (in this case) recover simultaneously both $RK^0[0,0]$ and $RK^1[3,0]$. That is, by trying all possible remaining 6 bits of $RK^0[0,0]$ and 8 bits of $RK^1[3,0]$, we predict the corresponding cache access of B'. The correct pair $(RK^0[0,0], RK^1[3,0])$ is found by comparing the predictions to the actual cache accesses, which are already validated by Figure 8. We use the information to validate that with 30 values of the targeted byte we can always recover the corresponding key bytes for the first and second rounds.

6 Discussion and Conclusions

Countermeasures. Multiple approaches for addressing cache attacks have been suggested. Randomization-based approaches [LWML16; WUG+19] decouple memory lines from cache sets. While such approaches may prevent an attacker from finding the cache sets that correspond to memory locations, they do not protect against attacks that rely on cache capacity [SKH+19], and vulnerabilities may remain.

Approaches that partition the cache [DJL+12; KPM12; LGY+16; SSCZ11; ZRZ16] prevent adversaries from manipulating the cache state of the victim's cache partition. These approaches often rely on the control the operating system has over the mapping of virtual addresses, and do not apply to the L1 cache we attack. To protect the L1 cache, it may be possible to prevent concurrent access to it and to flush its contents upon context switch [GYCH19]. However, as Ge et al. [GYH18] note, it may be impossible to completely eliminate hardware leakage in current processors.

Ensuring that cache lines are accessed in a secret-independent order [BGNS06] can prevent our attack, but may still be susceptible to attacks that have sub-cache-line

resolution [IMB+19a; YGH16]. Preventing secret-dependent memory access using constant-time programming [BLS12] is considered a secure approach for protecting cryptographic implementations. When applied to Pilsung, using this approach would require reading all of the values in an S-Box when it is applied. Fortunately for Pilsung, during each round it accesses each of the 160 S-Boxes exactly once. Thus, scanning each of these tables results in a sequential memory access pattern throughout the memory allocated for the S-Boxes. Luckily, such access patterns tend to be highly optimized in modern processors, and the optimization is likely to mask some of the added overhead of scanning the whole S-Box. We leave the task of implementing this countermeasure and benchmarking its performance to future work.

On the use of key dependent S-Boxes. Interestingly, the use of key-dependent S-Boxes weakens the security of the overall implementation against cache-based side-channel attacks. This is since, in the implementation analyzed in this paper, each of the 160 S-Boxes are stored in a different part of the memory where accesses to them trigger different cache sets. Thus, by observing a specific cache set the attacker gets more precise information about specific bytes of the secret key. As each Pilsung S-Box consists of the original AES S-Box with an additional key-dependent permutation on the output bits, the implementation can only store one copy of the AES S-Box and compute the permutations online. This would avoid releasing the information on which S-Box is accessed, but would result in a cost to performance. Moreover, even the single copy of the AES S-Box can be calculated online using the algebraic equations used for its construction, effectively eliminating any microarchitectural leakage.

Limitations. The main focus of this work is to demonstrate and to analyze the weakness of the Pilsung implementation as reverse-engineered in [Kry18b]. This results in a few limitations of the attack presented in this paper. First, we trust the correctness of the reverse-engineering effort and that the source indeed matches the original Pilsung code and specifications. While we have no reason to doubt the correctness of the published code, in case of a mismatch between the published code and the original cipher our attack only applies to the code of [Kry18b].

The cache attack we demonstrate makes several assumptions that render it impractical for a real attack scenario. First, our attack is synchronous, implying that the attacker has the ability to trigger cipher invocations at precise times (probing the cache between invocations). While synchronous attacks are common in the literature [IAES15; OST06; WHS12; ZW10], we acknowledge that they represent a weaker attack scenario. However, we note that the gap between weak attack scenarios, such as ours, and more realistic attacks has been researched in the past [OST06]. As such, we are confident that the attack could apply in more realistic scenarios, albeit with more noise and consequently more samples.

Finally, our attack targets the L1 cache, which implies that the attacker and victim must be located on the same core. We note that performing a Prime+Probe attack on the last-level cache (LLC) attacks is a solved problem [IAES15; LYG+15]. Furthermore, we note that these attacks offer new opportunities for attacks on Pilsung. The S-Boxes of the different rounds overlap in the sets of the L1 cache that we target. Consequently, we need to take multiple samples to identify the access in the round we target. The number of cache sets of the LLC is significantly larger, and thus the likelihood that S-Boxes overlap is much smaller, potentially allowing the attacker to distinguish between accesses in different rounds. We leave the investigation of this scenario for future work.

Conclusions. In this work we investigate the reverse-engineered implementation of the North Korean cipher Pilsung. We show that using key-dependent S-Boxes and permutations

does not provide an adequate protection against cache attacks. We analyze the cipher and demonstrate complete key recovery after observing 35 million encryptions. The complete attack takes an average of eight minutes. Thus, our work demonstrates once again the need for careful, constant-time implementation as a defense for side-channel attacks.

Acknowledgements

We would like to thank our shepherd, Dan Page, and the anonymous reviewers for the constructive feedback.

This research was partially supported by gifts from Intel and from AMD and by the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-19-C-0531. This research was also partially supported by the Singapore National Research Foundation under its National Cybersecurity R&D Grant ("Cyber-Hardware Forensics & Assurance Evaluation R&D Programme" grant NRF2018NCR–NCR009–0001).

References

[AARR02] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. "The EM side-channel(s)". In: *CHES*. 2002, pp. 29–45.

[AK06] Onur Acıiçmez and Çetin Kaya Koç. "Trace-Driven Cache Attacks on AES (Short Paper)". In: *ICICS*. 2006, pp. 112–121.

[BBG+17] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. "Sliding Right into Disaster: Left-to-Right Sliding Windows Leak". In: *CHES*. 2017, pp. 555–576.

[Ber05] Daniel J. Bernstein. *Cache-timing attacks on AES*. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf. 2005.

[BGNS06] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. *Software mitigations to hedge AES against cache-based software side channel vulnerabilities*. IACR ePrint 2006/52. 2006.

[BH09] Billy Bob Brumley and Risto M. Hakala. "Cache-Timing Template Attacks". In: *ASIACRYPT*. 2009, pp. 667–684.

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. "The Security Impact of a New Cryptographic Library". In: *LATINCRYPT*. 2012, pp. 159–176.

[BM06] Joseph Bonneau and Ilya Mironov. "Cache-Collision Timing Attacks Against AES". In: *CHES*. 2006, pp. 201–215.

[BMD+17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *WOOT*. 2017.

[BPSY14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. ""Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way". In: *CHES*. 2014, pp. 75–92.

[CABH+19] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. "Port Contention for Fun and Profit". In: *IEEE SP*. 2019, pp. 1036–1053.

[DDME+18] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. "CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 171–191.

[DJL+12] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. “Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks”. In: *TACO* 8.4 (2012), 35:1–35:21.

[Fip] *FIPS 197: Specification for the Advanced Encryption Standard (AES)*. NIST. 2001.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. “Cache Games – Bringing Access-Based Cache Attacks on AES to Practice”. In: *IEEE SP*. 2011, pp. 490–505.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. “Electromagnetic analysis: Concrete results”. In: *CHES*. 2001, pp. 251–261.

[GOC16] Jose Javier Gonzalez Ortiz and Kevin J. Compton. *A Simple Power Analysis Attack on the Twofish Key Schedule*. CORR arXiv:1611.07109. 2016.

[GPTY18] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. “Drive-By Key-Extraction Cache Attacks from Portable Code”. In: *ACNS*. 2018, pp. 83–102.

[GRBG18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”. In: *USENIX Security*. 2018, pp. 955–972.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security*. 2015, pp. 897–912.

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. “RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis”. In: *CRYPTO (1)*. 2014, pp. 444–461.

[GVY17] Daniel Genkin, Luke Valenta, and Yuval Yarom. “May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519”. In: *ACM CCS*. 2017, pp. 845–858.

[GYCH18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware”. In: *J. Cryptographic Engineering* 8.1 (2018), pp. 1–27.

[GYCH19] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. “Time Protection: The Missing OS Abstraction”. In: *EuroSys*. 2019, 1:1–1:17.

[GYH18] Qian Ge, Yuval Yarom, and Gernot Heiser. “No Security Without Time Protection: We Need a New Hardware-Software Contract”. In: *APSys*. 2018, 1:1–1:9.

[IAES15] Gorka Irazoqui Apecechea, 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: *IEEE SP*. 2015, pp. 591–604.

[IAIES14] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. “Wait a Minute! A fast, Cross-VM Attack on AES”. In: *RAID*. 2014, pp. 299–319.

[IMB+19a] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. “SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks”. In: *USENIX Security*. 2019, pp. 621–637.

[IMB+19b] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks". In: *USENIX Security*. 2019, pp. 621–637.

[KJJ98] Paul Kocher, Joshua Jaffe, and Benjamin Jun. *Introduction to differential power analysis and related attacks*. https://www.rambus.com/wp-content/uploads/2015/08/DPATechInfo.pdf. 1998.

[Koc96] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *CRYPTO*. 1996, pp. 104–113.

[KPM12] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. "STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud". In: *USENIX Security*. 2012, pp. 189–204.

[LGY+16] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. "CATalyst: Defeating last-level cache side channel attacks in cloud computing". In: *HPCA*. 2016, pp. 406–418.

[LWML16] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. "Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks". In: *IEEE Micro* 36.5 (2016), pp. 8–16.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: *IEEE SP*. 2015, pp. 605–622.

[MCS17] Chujiao Ma, John A. Chandy, and Zhijie Shi. "Algebraic Side-Channel Attack on Twofish". In: *J. Internet Serv. Inf. Secur.* 7.2 (2017), pp. 32–43.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". In: *CHES*. 2017, pp. 69–90.

[NRMW12] Phuong Ha Nguyen, Chester Rebeiro, Debdeep Mukhopadhyay, and Huaxiong Wang. "Improved Differential Cache Attacks on SMS4". In: *Inscrypt*. 2012, pp. 29–45.

[NS06] Michael Neve and Jean-Pierre Seifert. "Advances on Access-Driven Cache Attacks on AES". In: *Selected Areas in Cryptography*. 2006, pp. 147–162.

[NSW06] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. "A refined look at Bernstein's AES side-channel analysis". In: *AsiaCCS*. 2006, p. 369.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *CT-RSA*. 2006, pp. 1–20.

[PBY17] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. "To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures". In: *ACM CCS*. 2017, pp. 1843–1855.

[PDR11] Rishabh Poddar, Amit Datta, and Chester Rebeiro. "A Cache Trace Attack on CAMELLIA". In: *InfoSecHiComNet*. 2011, pp. 144–156.

[Per05] Colin Percival. "Cache Missing for Fun and Profit". In: *BSDCan 2005*. 2005. URL: http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf.

[PGBY16] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. ""Make Sure DSA Signing Exponentiations Really are Constant-Time"". In: *ACM CCS*. 2016, pp. 1639–1650.

[PGM+16] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *USENIX Security*. 2016, pp. 565–581.

[QS01] Jean-Jacques Quisquater and David Samyde. "Electromagnetic analysis (EMA): Measures and counter-measures for smart cards". In: *CARDIS*. 2001, pp. 200–210.

[RGG+19] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. "The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations". In: *IEEE SP*. 2019, pp. 435–452.

[RM11] Chester Rebeiro and Debdeep Mukhopadhyay. "Cryptanalysis of CLEFIA Using Differential Methods with Cache Trace Patterns". In: *CT-RSA*. 2011, pp. 89–103.

[RMTF09] Chester Rebeiro, Debdeep Mukhopadhyay, Junko Takahashi, and Toshinori Fukunaga. "Cache Timing Attacks on Clefia". In: *INDOCRYPT*. 2009, pp. 104–118.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds". In: *ACM CCS*. 2009, pp. 199–212.

[SKH+19] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. "Robust Website Fingerprinting Through the Cache Occupancy Channel". In: *USENIX Security*. 2019.

[SNK+13] Alexander Schlösser, Dmitry Nedospasov, Juliane Krämer, Susanna Orlic, and Jean-Pierre Seifert. "Simple photonic emission analysis of AES". In: *J. Cryptographic Engineering* 3.1 (2013), pp. 3–15.

[SP13] Raphael Spreitzer and Thomas Plos. "Cache-Access Pattern Attack on Disaligned AES T-Tables". In: *COSADE*. 2013, pp. 200–214.

[SSCZ11] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring". In: *DSN Workshops*. 2011, pp. 194–199.

[TSS+03] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. "Cryptanalysis of DES Implemented on Computers with Cache". In: *CHES*. 2003, pp. 62–76.

[TTMH02] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Hiyauchi. "Cryptanalysis of Block Ciphers Implemented on Computers with Cache". In: *International Symposium on Information Theory and Its Applications*. 2002.

[WHS12] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. "A Cache Timing Attack on AES in Virtualization Environments". In: *Financial Cryptography*. 2012.

[WUG+19] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization". In: *USENIX Security*. 2019.

[Yar16] Yuval Yarom. *Mastik: A Micro-Architectural Side-Channel Toolkit*. http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf. Sept. 2016.

[YF14] Yuval Yarom and Katrina Falkner. "Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security*. 2014, pp. 719–732.

[YFT18] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. "Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures". In: *CoRR* abs/1808.04761 (2018).

[YGH16] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA". In: *CHES*. 2016, pp. 346–367.

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-VM side channels and their use to extract private keys". In: *ACM CCS*. 2012, pp. 305–316.

[ZRZ16] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. "A Software Approach to Defeating Side Channels in Last-Level Caches". In: *ACM CCS*. 2016, pp. 871–882.

[ZW10] Xin-jie Zhao and Tao Wang. *Improved Cache Trace Attack on AES and CLEFIA by Considering Cache Miss and S-box Misalignment*. IACR Cryptology ePrint Archive 2010/56. 2010.

[ZWZ09] Xin-jie Zhao, Tao Wang, and Yuanyuan Zheng. *Cache Timing Attacks on Camellia Block Cipher*. Cryptology ePrint Archive, Report 2009/354. 2009.

[Kry18a] Kryptos Logic. *A Brief Look At North Korean Cryptography*. https://www.kryptoslogic.com/blog/2018/07/a-brief-look-at-north-korean-cryptography/. July 2018.

[Kry18b] Kryptos Logic. *pilsung.c*. https://blog.kryptoslogic.com/assets/pyongyang/pilsung.c. July 2018.