Side-Channeling the Kalyna Key Expansion

Chitchanok Chuengsatiansup[1], Daniel Genkin[2],
Yuval Yarom[1], and Zhiyuan Zhang[1]

[1] University of Adelaide
[2] Georgia Institute of Technology

Abstract. In 2015, the block cipher Kalyna has been approved as the new encryption standard of Ukraine. The cipher is a substitution-permutation network, whose design is based on AES, but includes several different features. Most notably, the key expansion in Kalyna is designed to resist recovering the master key from the round keys.

In this paper we present a cache attack on the Kalyna key expansion algorithm. Our attack observes the cache access pattern during key expansion, and uses the obtained information together with one round key to completely recover the master key. We analyze all five parameter sets of Kalyna. Our attack significantly reduces the attack cost and is practical for the Kalyna-128/128 variant, where it is successful for over 97% of the keys and has a complexity of only $2^{43.58}$. To the best of our knowledge, this is the first attack on the Kalyna key expansion algorithm.

To show that the attack is feasible, we run the cache attack on the reference implementation of Kalyna-128/128, demonstrating that we can obtain the required side-channel information. We further perform the key-recovery step on our university's high-performance compute cluster. We find the correct key within 37 hours and note that the attack requires 50 K CPU hours for enumerating all key candidates.

As a secondary contribution we observe that the additive key whitening used in Kalyna facilitates first round cache attacks. Specifically, we design an attack that can recover the full first round key with only seven adaptively chosen plaintexts.

1 Introduction

Since the seminal work of Kocher [32], side-channel attacks have become a major threat to the security of virtually any cryptographic primitive. While side channels come in many forms and exploitation techniques [33,40,67], a particular recent concern is microarchitectural side channels [21], which extract secret information by exploiting variations in instruction timing due to contention on CPU resources. Since the introduction of cache attacks [47,48], it appears that nearly every microarchitectural feature in modern CPUs can be used to attack cryptographic primitives across hardware-backed security boundaries [1,2,12,14,47,66].

With respect to targeted cryptographic protocols, most research attention have been given to Western standards such as AES [8,29,30,42,47], RSA [7,13,37,64], elliptic-curve cryptography [4,49], presumably due to their wide spread adoption, readily-available standardization documents, and acceptability of reference implementations. Much less attention in comparison has been given to national cipher standards of former Eastern Bloc countries, which often deploy their own cryptography standards [6,34,45,54]. While modern symmetric Eastern Bloc designs are often based on AES, these often include variations in features such as round function and key expansion, due to different trade-offs between security and performance made by local standardization agencies while adapting the cipher to local use. As side-channel attacks are by their very nature implementation specific, it is unclear how such local adaptations impact the cipher's side-channel resistance.

1.1 Our Contribution

Tackling this issue, in this work we investigate the Ukrainian cipher Kalyna [45] as a case study of an Eastern Bloc cipher. At a high level, Kalyna is modeled after AES, but has some important differences. First, Kalyna's key expansion algorithm is considered to be non-reversible [3,20]. Thus, a side-channel adversary that wishes to attack Kalyna needs to retrieve all of the round keys. Moreover, Kalyna uses an arithmetic addition operation, which is non-linear in GF(2), for pre- and post-whitening. This is known to hinder cryptanalysis [35,43].

Despite these changes, we present the first cache attack against the Kalyna key expansion algorithm, which significantly reduces the attack complexity in all five variants (see Table 1). We also show that an attacker can practically extract the secret key from Kalyna-128/128's reference implementation with a 97% success probability and an expected complexity of $2^{43.58}$. We demonstrate that Kalyna's computation of the round keys from the master key propagates differences between internal values in a way that allows using them for recovering the master key. To the best of our knowledge, this is the first attack that recovers the master key of Kalyna.

We further demonstrate that the additive key whitening used in Kalyna is more vulnerable to cache attacks than the more common Boolean whitening. The main cause is that due to carry ripples during addition, an attacker can use a simple binary search to find the key. The attack requires only seven (chosen) plaintexts to completely recover the key.

Table 1: Attack cost

Variant	Block size	Key length	Rounds	Columns	Attack cost
Kalyna-128/128	128	128	10	2	$2^{43.58}$
Kalyna-128/256	128	256	14	2	$2^{80.47}$
Kalyna-256/256	256	256	14	4	$2^{168.60}$
Kalyna-256/512	256	512	18	4	$2^{173.45}$
Kalyna-512/512	512	512	18	8	$2^{363.27}$

1.2 Paper Outline

The rest of this paper is organized as follows. Section 3 presents an overview of the attack and describes the information obtained through the side channel. The details of the attack on Kalyna-128/128 is described in Section 4, followed by a description of the attacks on other variants of Kalyna in Section 5. In Section 6 we describe our implementations of proof-of-concept attacks, showing that both the cache side-channel attack and key recovery are feasible.

2 Background

In this section we present background on the Kalyna cipher and the notation we use to describe it, cache attacks, and the related work.

2.1 Kalyna

Kalyna is a substitution-permutation network whose design is based on AES. It was one of the proposals for the Ukrainian National Public Cryptographic Competition which was held between 2007 and 2010. In 2015, the cipher was approved as the national standard of Ukraine.

Kalyna has five variants offering different key and block sizes as summarized in Table 1. We mainly focus on the Kalyna-128/128 variant. For further details and for other variants, see [45].

Bytes. The basic unit of information that Kalyna uses is a byte, which, following standard convention, is a sequence of eight bits each having the value zero or one. We use subscripts to refer to specific bits of a byte. Thus, a byte b consists of the bits $b_0, \ldots, b_7$. Depending on context, a byte represents one of three entities: a sequence of bits, a number in the range $[0, \ldots, 255]$ where the byte b represents $\sum b_i 2^i$, or an element in the field of polynomials modulo $x^8 + x^4 + x^3 + x^2 + 1$ over $\mathrm{GF}(2^8)$.

We use the $+$ and $\cdot$ operations for addition and multiplication of numbers. Symbol $\oplus$ is used for polynomial addition and also doubles as bitwise XOR (exclusive or). We denote the bitwise and operation with &.

Matrices. Kalyna represents most of its data as matrices of bytes. These matrices have eight rows. The number of columns varies, but most matrices in Kalyna-128/128 have two columns. For a matrix M we use $M[i]$ to refer to the i^{th} byte, in column-first order, i.e. $M[0]$ is the first byte of the first column, $M[1]$ is the second byte of the first column, and $M[8]$ is the first byte of the second column. We use $M[i, \ldots, j]$ to denote a subsequence of bytes in a matrix. Columns can be interpreted both as sequences of bytes and as numbers modulo 2^{64}, where $M[8i, \ldots, 8i+7]$ represents the number $\sum_{j=0}^{7} 256^j M[8i+j]$. Given a set of indices Q, the projection of a matrix M over Q, denoted by $M|_Q$ is the matrix M where all indices not in Q are zeroed. That is:

$$M|_Q[i] = \begin{cases} M[i] & i \in Q \\ 0 & i \notin Q \end{cases}$$

We sometimes use a graphical notation to better highlight the indices selected by the set of indices. Specifically, we draw a matrix, with filled rectangles representing the selected indices. Thus M selects the even indices of M, i.e. $M|_{\{2i|0\leq i<8\}}$, and M $= M|_{\{5,7,9,11\}}$.

Kalyna supports two addition operations on matrices. Matrix (bitwise) addition, denoted with $\oplus$, designates the conventional operation over the polynomial field. Matrix column addition modulo 2^{64}, denoted by $\boxplus$, produces an output matrix by adding columns. Specifically, the operator treats each column of the input matrices as a 64-bit number. To produce a column of the output matrix, the operator adds the two corresponding input columns (modulo 2^{64}) and interprets the result of the sum as a matrix column. Crucially, unlike regular addition, in column addition modulo 2^{64} carries propagate between the cells of same matrix column.

More formally, let $M[8i,\ldots,8i{+}7] = \sum_{j=0}^{7} 256^j M[8i{+}j]$ and $N[8i,\ldots,8i{+}7] = \sum_{j=0}^{7} 256^j N[8i{+}j]$, for matrices M and N we have:

$$\begin{aligned}(M \boxplus N)[8i,\ldots,8i+7] &= M[8i,\ldots,8i+7] + N[8i,\ldots,8i+7] \\ &= \left(\sum_{j=0}^{7} 256^j M[8i+j] + \sum_{j=0}^{7} 256^j N[8i+j]\right) \bmod 2^{64}.\end{aligned}$$

Moreover, we use $\boxminus$ to denote the matrix column subtraction modulo 2^{64}. Finally, we use $\Sigma(M)$ to denote the sum of the columns of the matrix M modulo 2^{64}. That is, $\Sigma(M) = \sum_i M[8i,\ldots,8i+7] = \sum_i \sum_{j=0}^{7} 256^j M[8i+j] \bmod 2^{64}$.

Cipher Structure. Kalyna follows the design of AES with multiple rounds. Each round consists of four operations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. Like AES, Kalyna also includes an initial plaintext whitening operation. Encryption starts by setting the initial *state* to the plaintext. The cipher then applies the operations to the state. The resulting state is the ciphertext. Decryption applies the operations in the reverse order.

SubBytes. Kalyna specifies four *substitution boxes* (S-Boxes), which provide the non-linear substitution step of each round. Each S-Box consists of 256 entries. Given a state S, the SubBytes step applies one of the S-Boxes to each of the state's byte. Specifically,

$$SB(S)[i] = SB_{i \bmod 4}[S[i]]$$

ShiftRows. As in AES, the ShiftRows moves bytes across rows in the state matrix. The transformation depends on the state size and thus varies with the different variants of the cipher. For Kalyna-128/128, the transform (see Figure 1) is defined as:

$$SR(S)[i] = \begin{cases} S[i] & i < 4 \\ S[i+8] & i \geq 4 \end{cases}$$

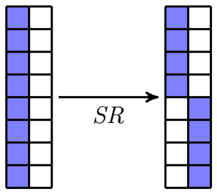


Fig. 1: The ShiftRows in Kalyna-128/128.

$$\begin{pmatrix} 1 & 1 & 5 & 1 & 8 & 6 & 7 & 4 \\ 4 & 1 & 1 & 5 & 1 & 8 & 6 & 7 \\ 7 & 4 & 1 & 1 & 5 & 1 & 8 & 6 \\ 6 & 7 & 4 & 1 & 1 & 5 & 1 & 8 \\ 8 & 6 & 7 & 4 & 1 & 1 & 5 & 1 \\ 1 & 8 & 6 & 7 & 4 & 1 & 1 & 5 \\ 5 & 1 & 8 & 6 & 7 & 4 & 1 & 1 \\ 1 & 5 & 1 & 8 & 6 & 7 & 4 & 1 \end{pmatrix}$$

Fig. 2: The MixColumns matrix.

MixColumns. The MixColumns transformation computes a linear function that mixes the values along the columns of the state. For Kalyna, MixColumns operates by multiplying the state matrix with the pre-defined 8-by-8 matrix (see Figure 2). An important property of the matrix is that it is maximum distance separable [38]. Consequently, given a total of eight out of sixteen known bytes in the inputs and outputs of the transformation, we can recover the missing eight bytes.

AddRoundKey. The AddRoundKey operation mixes key material into the state. Kalyna expands the master key K to multiple *round keys*, where round key RK_i is used in the i^{th} round. In each of the first nine rounds in Kalyna-128/128, AddRoundKey uses matrix addition, i.e. $ARK(S, RK_i) = S \oplus RK_i$. The 10^{th} round (last round) uses column addition modulo 2^{64}, i.e. $ARK(S, RK_{10}) = S \boxplus RK_{10}$. Furthermore, Kalyna has a key whitening step before the first round, where it uses column addition modulo 2^{64} to mix additional round key, RK_0, with the plaintext.

Key Expansion. Unlike AES, Kalyna uses a complex, non-reversible procedure for generating the round keys. The procedure first generates an intermediate key K_σ. This K_σ is then combined with the master key K to generate the *even* round keys. Figure 3 shows the process.

Generating K_σ. As the left side of Figure 3 shows, to generate the intermediate key K_σ, Kalyna performs three sets of SubBytes, ShiftRows and MixColumns operations. The inputs of this process are the master key K and a constant C that depends on the Kalyna variant. Kalyna-128/128 uses the constant 5, i.e. a matrix having all bytes zero except for the least significant byte whose value is 5.

Generating RK_i. Generating the round keys RK_i follows a similar procedure, but with two sets of SubBytes, ShiftRows and MixColumns operations, see Figure 3 (right). For Kalyna-128/128, one of the input is K_i which is either the master key K, if $i \bmod 4$ is 0 or 1, or the master key with the two columns swapped, if $i \bmod 4$ is 2 or 3. More formally, we have:

$$K_i[j] = \begin{cases} K[j] & \text{if } \lfloor i/2 \rfloor \text{ is even} \\ K[j \oplus 8] & \text{if } \lfloor i/2 \rfloor \text{ is odd} \end{cases}$$

Another input is the intermediate key K_σ modified by column addition modulo 2^{64} with a round constant C_i. The round constant for round i is a matrix with the value 0 in odd indices, and $2^{\lfloor i/2 \rfloor}$ in even indices.

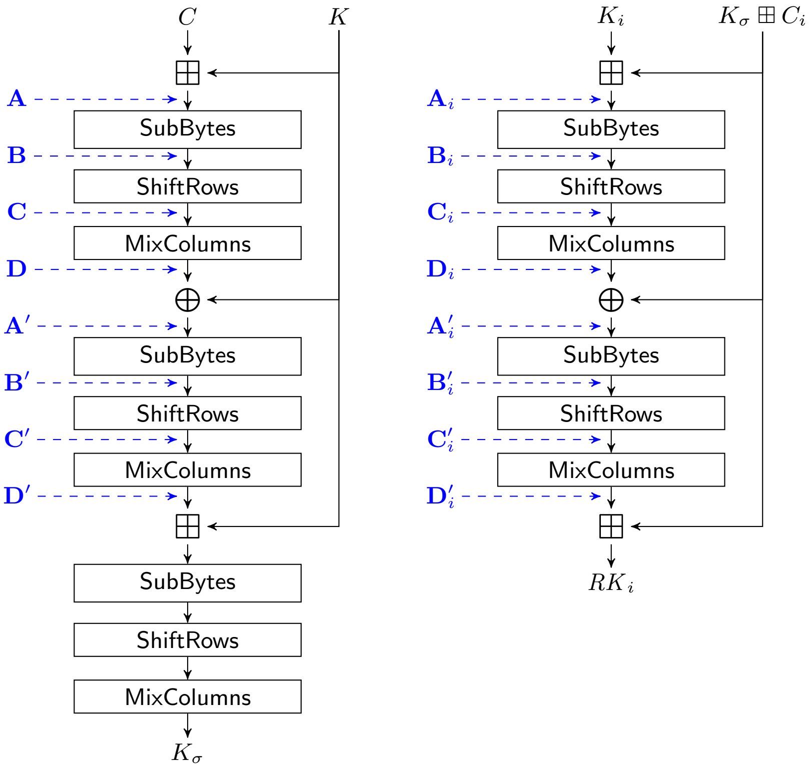


Fig. 3: The Kalyna-128/128 key expansion. Notations such as $\mathbf{A}, \mathbf{A}', \mathbf{A}_i, \mathbf{A}'_i$ are used as explicit reference to internal values computed during the key expansion.

Notation. To facilitate referencing the intermediate states during the generation of the round keys, Figure 3 also contains explicit names of various intermediate states. For instance, State $\mathbf{A}_0$ is the result of column addition modulo 2^{64} of K_0 and $K_\sigma \boxplus C_0$, i.e. $\mathbf{A}_0 = K_0 \boxplus K_\sigma \boxplus C_0$, $\mathbf{B}_0$ is the result of applying SubBytes to $\mathbf{A}_0$. Hence, $\mathbf{B}_0 = SB(K_0 \boxplus K_\sigma \boxplus C_0)$.

2.2 Cache Attacks

Caches. Caches are small and fast banks of memory that bridge the speed gap between the fast processor and the slower memory by exploiting the temporal and spatial locality that software exhibits. More specifically, the entire memory space is divided into fixed-size *lines*, typically of size 64 bytes. When the processor needs to access memory, it first checks if the required line resides in the cache. In case of *cache hit*, when the required line is in the cache, the memory access request is served from the cache. Conversely, in a *cache miss*, when the required line is not in the cache, the processor is forced to retrieve the line from the slower main memory, storing a copy in the cache for potential future use. Typically, due to the limited size of the cache, the processor needs to *evict* another line from the cache to make room for storing the retrieved cache line.

Set-associative caches. Modern caches are often *set-associative*. The cache is divided into multiple *sets*, each containing a fixed number of *ways*. Each memory block is mapped to a single cache set and can only be stored in the set it maps to. Vendors do not always publish the details of the mapping function; however past research has shown that the function can be reverse engineered, allowing a user to determine the cache set that stores a given memory block [27,28,41,65].

Cache-based side-channels. Because caches are typically shared between multiple programs, a malicious program that monitors the cache can learn information on the execution of other programs. This can be used to leak sensitive information across security-domain boundaries. Over the years, many cache attacks have been designed, demonstrating retrieval of encryption keys [13,15,17, 22,23,26,31,37,42,47,50,64] as well as other sensitive information [24,25,55,62].

Prime+Probe. Prime+Probe [37,47] is a cache attack technique that exploits the set-associative structure of the cache. In the *Prime* phase, the attacker completely fills one or more cache sets with its data. The attacker then waits, letting the victim execute for a certain duration. As the cache is already full, any memory access performed by the victim during its execution must cause eviction of the attacker's data back into the machine's main memory. Finally, in the *Probe* phase, the attacker measures the time to access the data previously used in the prime phase to fill the machine's cache. A short access time indicates that the data is still cached, whereas long access time indicates that data has been evicted. Thus, the attacker learns which cache sets the victim has accessed between the prime and the probe phases, exploiting the mapping between cache sets and address bits to recover which address the victim has accessed.

Temporal resolution. The time to execute one round of Prime+Probe depends on the specific cache and the number of cache sets monitored, ranging from a few thousands of cycles [37] and up to millions [55]. Past research has demonstrated several techniques for improving the temporal resolution of observed events. These include exploiting the OS scheduler to frequently interrupt the victim [14, 26, 31], degrading the performance of the victim by contending on resources the victim requires [4, 7, 11, 50, 51], and exploiting elevated privileges [9, 10, 17, 42, 56].

2.3 Related Work

A few side-channel attacks on Kalyna have been published. Fernbandes Medeiros et al. [20] propose a correlation power analysis on Kalyna-128/128 where the attack recovers all round keys with the success rate of 96% using 250 measured values. Later on, Duman and Youssef [19] propose fault attacks on Kalyna. They employ differential fault analysis and ineffective fault analysis to recover round keys. Their attack works by reducing the number possible candidates then bruteforce to find the correct ones. To the best of our knowledge, no cache attacks on Kalyna have been published.

There also exist differential cryptanalysis on reduced-round Kalyna where most of the focus is on the variants whose key length is double the block size. Akshima et al. [3] use meet-in-the-middle attack to recover subkeys for 9-round Kalyna-128/256 and Kalyna-256/512. In [5], the authors recover all round keys using parameters matching. Later on, [59] improve the attack on Kalyna-128/256 by using more optimal differential paths. Similarly, [36] propose a chosen plaintext, reduced-round Kalyna attack on Kalyna-128/256 and Kalyna-256/512 where the attack recovers the round keys.

In addition to differential cryptanalysis attack, the recent work [35] employs an impossible differential attack, i.e. analysing input difference which never results in a particular output difference, on all variants of Kalyna. Nevertheless, with the reduced-round attacks, the complexity is still relatively high. For example, the attack on 4-round Kalyna-128/128 requires 2^{103} time complexity.

Note that all those attacks, only recover the round keys. Kalyna's key expansion prevents using information from one or more round keys to derive the missing keys or the master key. Hence, those attacks are forced to recover all of the (reduced) round keys to be able to encrypt/decrypt messages. No published attacks have recovered the master key.

Twofish [52,53] is a block cipher that also uses a non-reversible key expansion procedure. Ortiz and Compton [46] demonstrate a power analysis attack on the key schedule of Twofish. Specifically, they target an 80-bit implementation of Twofish for a smartcard and show that the key can be recovered from the power traces even in the presence of errors. Attack on key schedules have also been demonstrated on DES [57,58], AES [18,39,60], and Serpent [16]. We note that the key schedule of these ciphers is not designed to deter recovery of the master key from the round keys.

3 Cryptanalysis Overview

Due to the complex key expansion algorithm in Kalyna, knowing one or even all of the round keys does not reveal the master key. Consequently, prior attacks on Kalyna focus on recovering the round keys. Instead, in this work we focus on recovering the master key in the presence of side-channel leakage from the key expansion algorithm. Specifically, we assume a side-channel oracle that reveals the two most significant bits (MSBs) of each S-Box access. More formally, given a byte b, the oracle $O(b)$ returns $64 \cdot \lfloor b/64 \rfloor$. In Section 6 we show how we realize such an oracle using the Prime+Probe [37,47] attack on the S-Box access.

Side-channel oracles tend to provide partial information on the observed state. To find the missing information, prior cache attacks on block ciphers combine the information observed over multiple inputs, which are typically assumed to be randomly chosen. In our attack, we do not have this option since we target the key expansion which is executed once (as opposed to the encryption which goes into multiple rounds) and the inputs used for key expansion are fixed and are not under attacker's control. Instead, our attack exploits the differences be-

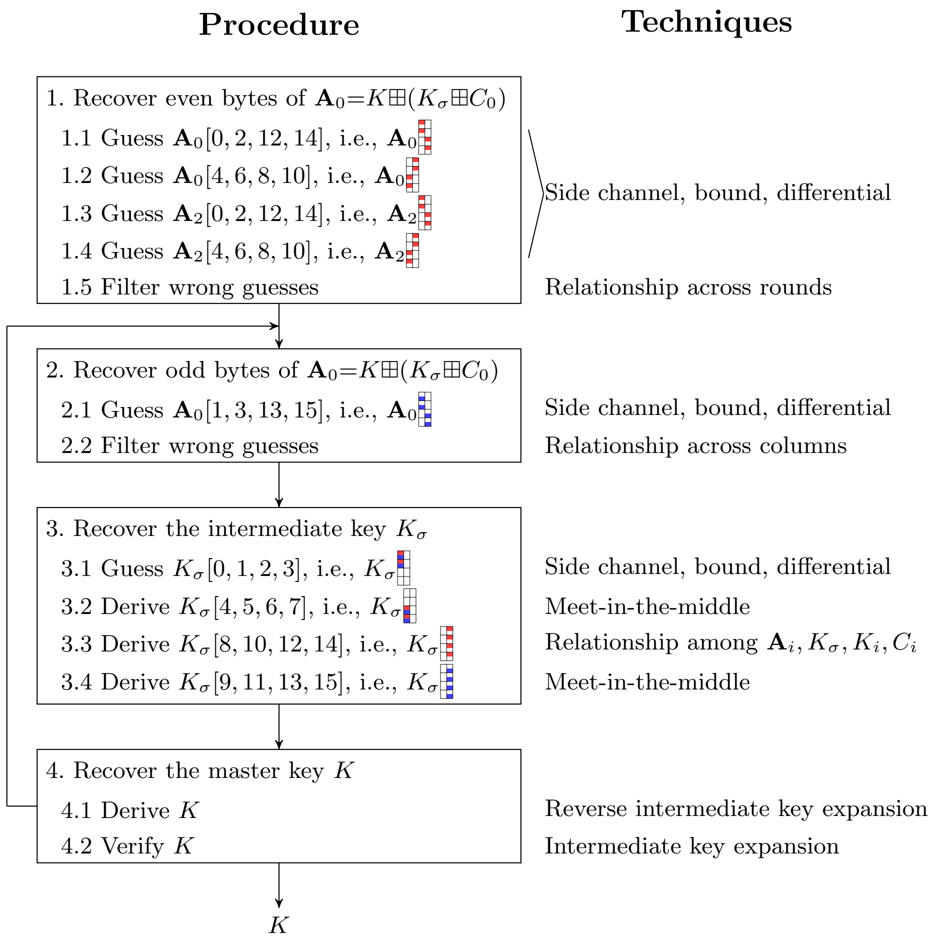

Procedure
Techniques
1. Recover even bytes of $\mathbf{A}_0 = K \boxplus (K_\sigma \boxplus C_0)$
1.1 Guess $\mathbf{A}_0[0, 2, 12, 14]$, i.e., $\mathbf{A}_0$
1.2 Guess $\mathbf{A}_0[4, 6, 8, 10]$, i.e., $\mathbf{A}_0$
1.3 Guess $\mathbf{A}_2[0, 2, 12, 14]$, i.e., $\mathbf{A}_2$
1.4 Guess $\mathbf{A}_2[4, 6, 8, 10]$, i.e., $\mathbf{A}_2$
1.5 Filter wrong guesses
Side channel, bound, differential
Relationship across rounds
2. Recover odd bytes of $\mathbf{A}_0 = K \boxplus (K_\sigma \boxplus C_0)$
2.1 Guess $\mathbf{A}_0[1, 3, 13, 15]$, i.e., $\mathbf{A}_0$
2.2 Filter wrong guesses
Side channel, bound, differential
Relationship across columns
3. Recover the intermediate key K_σ
3.1 Guess $K_\sigma[0, 1, 2, 3]$, i.e., K_σ
3.2 Derive $K_\sigma[4, 5, 6, 7]$, i.e., K_σ
3.3 Derive $K_\sigma[8, 10, 12, 14]$, i.e., K_σ
3.4 Derive $K_\sigma[9, 11, 13, 15]$, i.e., K_σ
Side channel, bound, differential
Meet-in-the-middle
Relationship among $\mathbf{A}_i, K_\sigma, K_i, C_i$
Meet-in-the-middle
4. Recover the master key K
4.1 Derive K
4.2 Verify K
Reverse intermediate key expansion
Intermediate key expansion
K

Fig. 4: Steps of our attacks and techniques used.

tween the inputs used for generating the different round keys and relationships between key parts to allow a more efficient search in the potential key space.

Our attack consists of four main steps as illustrate in Figure 4 for Kalyna-128/128. We first use the structure of the key schedule algorithm to guess the value of $\mathbf{A}_0$, we then use a meet-in-the-middle attack to find the value of K_σ, which allows us to recover K.

4 Attacking Kalyna-128/128

In this section, we present the attack on the smallest variant of the cipher, Kalyna-128/128. Section 5 presents an outline of the differences due to the increase in the block size or the key in other variants of the cipher.

4.1 Recover Even Bytes of $\mathbf{A_0}$

The first step in recovering the master key K is to recover the even byte of $\mathbf{A}_0 = K \boxplus (K_\sigma \boxplus C_0)$. Naively, there are 2^{64} possible candidates for the even bytes of $\mathbf{A}_0$. This section explain how we reduce the search space and narrow down to only 1.4 candidates on average. The main tool is the side-channel information from the oracle revealing the two MSBs of each SubBytes access. This enables us to set the bound of the search and verify the correctness of the guesses.

According to the description of the Kalyna cipher [45], we observe that $K_0 = K_4 = K_8$ and $K_2 = K_6 = K_{10}$. Consequently, we have $\mathbf{A}_4 = \mathbf{A}_0 \boxplus (C_4 \boxminus C_0)$ where $C_4 \boxminus C_0$ is a known constant with the value 3 for even bytes and 0 for odd bytes. Depending on the value of the LSBs of $\mathbf{A}_0[0]$, adding 3 may or may not cause a change in the two MSBs, i.e. the value returned from the side-channel oracle $O(\cdot)$. For example, $O(\mathbf{A}_0[0]) \neq O(\mathbf{A}_4[0])$ indicates that the value in the six LSBs of $\mathbf{A}_0[0]$ is between 61 and 63. Otherwise, it is below 61. By comparing $O(\mathbf{A}_0)$ to $O(\mathbf{A}_4)$ we can tighten the bounds on possible values of the six LSBs of $\mathbf{A}_0$. This also applies to $\mathbf{A}_2$ by considering $O(\mathbf{A}_2), O(\mathbf{A}_6)$ and $O(\mathbf{A}_{10})$. Table 2 shows the bounds for the six LSBs of even bytes of $\mathbf{A}_0$ and $\mathbf{A}_2$ in Kalyna-128/128. Note that the table ignores carries from odd to even bytes because they are rare. They only make minor changes to the bounds and even make the attack easier since they reveal a significant amount of information about the key.

Table 2: Bounds on LSBs of even bytes of $\mathbf{A}_0$ and $\mathbf{A}_2$ in Kalyna-128/128.

Condition	Range for $\mathbf{A}_0[j]$	Range for $\mathbf{A}_2[j]$
$O(\mathbf{A}_i[j]) = O(\mathbf{A}_{i+8}[j])$	0 – 48	0 – 33
$O(\mathbf{A}_{i+4}[j]) \neq O(\mathbf{A}_{i+8}[j])$	49 – 60	34 – 57
$O(\mathbf{A}_i[j]) \neq O(\mathbf{A}_{i+4}[j])$	61 – 63	58 – 63

Recall that with a high probability, there is only a small difference in the even bytes of $\mathbf{A}_0$, $\mathbf{A}_4$, and $\mathbf{A}_8$ (resp. $\mathbf{A}_2$, $\mathbf{A}_6$, and $\mathbf{A}_{10}$) while the odd bytes are identical. We initially observe the propagated differences among round key expansions at $\mathbf{A}'_0$, $\mathbf{A}'_4$, and $\mathbf{A}'_8$ under two simplifying assumptions.

Assumption 1. Overflows of even bytes when adding C_i to $K_i \boxplus K_\sigma$ do not depend on i. This implies that for odd j, $\mathbf{A}_i[j] = \mathbf{A}_{i+4}[j]$, i.e. $\mathbf{A}_0$▤ $=$ $\mathbf{A}_4$▤ $=$ $\mathbf{A}_8$▤ and $\mathbf{A}_2$▤ $=$ $\mathbf{A}_6$▤ $=$ $\mathbf{A}_{10}$▤.

Assumption 2. When adding C_i to K_σ, the carry from bit five to bit six of even bytes does not depend on i. This implies that $O((K_\sigma \boxplus C_i)[j]) = O((K_\sigma \boxplus C_{i+4})[j])$ for all $0 \leq j < 16$. We omit the index j when we refer to all the 16 bytes.

We can guess some of the even bytes of $\mathbf{A}_0$, determining the corresponding bytes in $\mathbf{A}_4$ and $\mathbf{A}_8$. We can then track how the differences between the bytes propagate through the first round of the round-key generation step to see how they affect $\mathbf{A}'_i$ and compare with the side-channel information we obtain on $\mathbf{A}'_i$.

We now look at the difference between oracle observations for $\mathbf{A}'_i$ and $\mathbf{A}'_{i+4}$.

$$\begin{aligned} O\left(\mathbf{A}'_i\right) \oplus O\left(\mathbf{A}'_{i+4}\right) &= O\left(\mathbf{A}'_i \oplus \mathbf{A}'_{i+4}\right) \\ &= O\left(\mathbf{D}_i \oplus (K_\sigma \boxplus C_i) \oplus \mathbf{D}_{i+4} \oplus (K_\sigma \boxplus C_{i+4})\right) \\ &= O\left(\mathbf{D}_i \oplus \mathbf{D}_{i+4}\right) \oplus O\left((K_\sigma \boxplus C_i) \oplus (K_\sigma \boxplus C_{i+4})\right) \qquad (1) \end{aligned}$$

Let Δ_i be the oracle difference between $K_\sigma \boxplus C_i$ and $K_\sigma \boxplus C_{i+4}$. That is,

$$\Delta_i = O\left((K_\sigma \boxplus C_i) \oplus (K_\sigma \boxplus C_{i+4})\right) = O\left(\mathbf{A}'_i\right) \oplus O\left(\mathbf{A}'_{i+4}\right) \oplus O\left(\mathbf{D}_i \oplus \mathbf{D}_{i+4}\right)$$

By Assumption 2, we have $\Delta_i = 0$. Hence,

$$O\left(\mathbf{A}'_i\right) \oplus O\left(\mathbf{A}'_{i+4}\right) = O\left(\mathbf{D}_i \oplus \mathbf{D}_{i+4}\right) \qquad (2)$$

We now look at each column of $\mathbf{A}'_i$ separately. For the first column, we have:

$$\begin{aligned} \mathbf{D}_i\text{▦} \oplus \mathbf{D}_{i+4}\text{▦} &= MC(\mathbf{C}_i\text{▦}) \oplus MC(\mathbf{C}_{i+4}\text{▦}) = MC(SR(\mathbf{B}_i\text{▦})) \oplus MC(SR(\mathbf{B}_{i+4}\text{▦})) \\ &= MC(SR(\mathbf{B}_i\text{▦} \oplus \mathbf{B}_i\text{▦})) \oplus MC(SR(\mathbf{B}_{i+4}\text{▦} \oplus \mathbf{B}_{i+4}\text{▦})) \\ &= MC(SR(SB(\mathbf{A}_i\text{▦}) \oplus SB(\mathbf{A}_{i+4}\text{▦}))) \oplus MC(SR(SB(\mathbf{A}_i\text{▦}) \oplus SB(\mathbf{A}_{i+4}\text{▦}))) \qquad (3) \end{aligned}$$

By Assumption 1 we have $SB(\mathbf{A}_i\text{▦}) = SB(\mathbf{A}_{i+4}\text{▦})$. Hence,

$$O(\mathbf{A}'_i) \oplus O(\mathbf{A}'_{i+4}) = O(MC(SR(SB(\mathbf{A}_i\text{▦}) \oplus SB(\mathbf{A}_{i+4}\text{▦})))) \qquad (4)$$

The side-channel observation provides the oracle values for the left-hand side of Equation 4. We can now guess $\widetilde{\mathbf{A}}_0$, the values of the four even bytes of $\mathbf{A}_0$▦, and calculate the corresponding $\widetilde{\mathbf{A}}_4 = \widetilde{\mathbf{A}}_0 \boxplus (C_4 \boxminus C_0)$, $\widetilde{\mathbf{D}}_0 = (MC(SR(SB(\widetilde{\mathbf{A}}_0))))$, and $\widetilde{\mathbf{D}}_4 = (MC(SR(SB(\widetilde{\mathbf{A}}_4))))$. If the guess is correct, i.e. when $\widetilde{\mathbf{A}}_0 = \mathbf{A}_0$▦, we will get

$$O\left(\mathbf{A}'_0\right)\text{▦} \oplus O\left(\mathbf{A}'_4\right)\text{▦} = O\left(\widetilde{\mathbf{D}}_0 \oplus \widetilde{\mathbf{D}}_4\right)\text{▦} \qquad (5)$$

If the guess is incorrect, the probability of a match is 2^{-16}. To see this, there is only one correct pattern of the 16 bits (two MSBs per byte of eight bytes).

Similarly, for $\widetilde{\mathbf{A}}_8 = \widetilde{\mathbf{A}}_0 \boxplus (C_8 \boxminus C_0)$, $\widetilde{\mathbf{D}}_8 = (MC(SR(SB(\widetilde{\mathbf{A}}_8))))$ we have

$$O\left(\mathbf{A}'_0\right)\text{▦} \oplus O\left(\mathbf{A}'_8\right)\text{▦} = O\left(\widetilde{\mathbf{D}}_0 \oplus \widetilde{\mathbf{D}}_8\right)\text{▦} \qquad (6)$$

with a probability 2^{-16} unless $\widetilde{\mathbf{A}}_0 = \mathbf{A}_0$▦.

With each byte having at most 49 possible values (c.f. Table 2), we need to examine $49^4 \approx 2^{22.5}$ possible combinations of values. The probability that the wrong guess matches both Equation 5 and Equation 6 is 2^{-32}. Hence, we expect that only the correct guess will match both.

Repeating the process for the right column of $\mathbf{A}'_0$ and for both columns of $\mathbf{A}'_2$, we can recover the even bytes of $\mathbf{A}_0$ and $\mathbf{A}_2$, at a complexity of less than $2^{22.5} \cdot 4 = 2^{24.5}$.

Handling Overflows in $K_\sigma \boxplus C_i$. The discussion so far makes two simplifying assumptions We now remove Assumption 2 and account for the possibility of carries from bit five to bit six of even bytes of $K_\sigma \boxplus C_i$. The main consequence is that Δ_i is not always zero and we need to accept some candidates when Equations 5 or 6 are not satisfied.

We now investigate the possible values in Δ_i. When an overflow occurs in an even byte $2j'$ for $0 \le j' < 8$, we have $O(K_\sigma \boxplus C_i)[2j'] = O(K_\sigma \boxplus C_{i+4})[2j'] + 64 \pmod{256}$. Hence, $\Delta_i[2j'] \in \{0\mathrm{x}40, 0\mathrm{xc}0\}$.

Under rare conditions, the overflow can percolate and affect the oracle of the following odd byte. Specifically, this can happen when an overflow occurs in byte $2j'$, the top two bits of the byte (before the overflow) are both set, and the six least significant bits of byte $2j'+1$ are also set. The probability of an overflow is less than 1/2, hence the probability of a change in the oracle of a given odd byte is lower than 2^{-9}, and the probability that this happens in any of the eight odd bytes of K_σ is less than 2%.

Thus, every byte of Δ_i can have three potential values: 0x00, 0x40, and 0xc0. However, if we naively accept all possible guesses where the oracle matches any of these values, we will accept incorrect guesses with a probability $2^{-3.32}$, which would leave a rather long list of candidates.[3] However, we observe that an overflow to an odd byte $2j'+1$ can only occur if $\Delta_i[2j'] = 0\mathrm{xc}0$ and that at most one of $\Delta_i[2j']$ and $\Delta_{i+4}[2j']$ can be non-zero. Thus, there are only nine possible value assignments for the tuple

$$\tau_{j'} = (\Delta_i[2j'], \Delta_i[2j'+1], \Delta_{i+4}[2j'], \Delta_{i+4}[2j'+1]). \tag{7}$$

The probability that an incorrect guess results in a possible combination of values, therefore, is $(9/256)^4 \approx 2^{-19.3}$. Hence, with an initial list of $2^{22.5}$ candidates, we expect that $2^{3.2} \approx 9$ incorrect guesses will remain.

Another strategy an attacker can adopt is to assume that overflows to odd bytes do not occur. This reduces the expected number of incorrect guesses to about 1, but also means that the attack will fail on some percentage of the keys.

Handling Overflows to Odd Bytes. We now handle the case that Assumption 1 does not hold. That is, when an odd byte $2j'+1$ changes between $\mathbf{A}_i$ and $\mathbf{A}_{i+4}$. This happens when the addition of C_i to $K_i \boxplus K_\sigma$ does not overflow byte $2j'$ whereas adding C_{i+4} does overflow the byte. We can detect such overflows by observing the oracle of $\mathbf{A}_i$ and $\mathbf{A}_{i+4}$. In the case of an overflow, we will have $O(\mathbf{A}_i)[2j'] = 0\mathrm{xc}0$ and $O(\mathbf{A}_{i+4})[2j'] = 0\mathrm{x}00$. This is in contrast with the case of the overflows of $K_\sigma \boxplus C_i$ discussed in the previous subsection, where we cannot observe overflows and need to guess them.

The main implication of overflows is that we can no longer split the even and odd bytes as in Section 4.1 and expect the part with the odd bytes to cancel out. To overcome this issue, we also guess odd bytes that we know change between key rounds, and adapt the split accordingly. For example, if we know that $\mathbf{A}_0[3] \neq \mathbf{A}_4[3]$, we get

$$\mathbf{D}_0 \oplus \mathbf{D}_4 = MC(SR(SB(\mathbf{A}_0) \oplus SB(\mathbf{A}_4))) \oplus MC(SR(SB(\mathbf{A}_0) \oplus SB(\mathbf{A}_4)))$$

[3] For each byte, the probability of accepting is 3/4. For eight bytes, it is $(3/4)^8 \approx 2^{-3.32}$

Having to guess odd bytes increases the number of guesses, affecting both the complexity of the step and the length of the list of potential guesses. In the worst case, when all the odd bytes in $\mathbf{A}_6$ differ from the corresponding bytes of $\mathbf{A}_{10}$, we need to guess four odd bytes at a complexity of 63^4 as well as four even bytes at a complexity of 24^4, giving a total complexity of $2^{42.3}$ (see Table 2). Note also that if the six LSBs of an odd byte $2j'+1$ are all set, we will observe that $O(\mathbf{A}_6[2j'+1]) \neq O(\mathbf{A}_{10}[2j'+1])$.

We can reduce the complexity of this case by first filtering the guesses based on $O(\mathbf{A}'_2) \oplus O(\mathbf{A}'_6)$ and only guess odd bytes for the surviving guesses. There is a total of $24^4 \approx 2^{18.3}$ guesses of even bytes. By the argument above, there are only five possible possible assignments for $\tau_{j'}$ (see Equation 7). Hence, the expected number of surviving guesses is $24^4 \cdot (5/16)^4 \approx 2^{11.6}$. Only for these surviving guesses we need to guess values of odd bytes, reducing the overall complexity to $63^4 \cdot 24^4 \cdot (5/16)^4 \approx 2^{35.5}$. While this approach reduces the search space, it does not reduce the expected number of guesses that survives the process, which remains at $63^4 \cdot 24^4 \cdot (9/256)^4 \approx 2^{22.9}$.

Another issue is that the subsequent steps of the attack make no use of the values of odd bytes of $\mathbf{A}_2$ and only limited use of odd bytes of $\mathbf{A}_0$. Consequently, when we need to guess more than two odd bytes in a column, the attack becomes significantly harder. In such a case, the maximum number of guesses for the two odd bytes is 63^2, for the two corresponding even bytes the number of guesses is no more than 24^2, and for the other two even bytes is up to 34^2. Thus, the total number of guesses is less than $63^2 \cdot 24^2 \cdot 34^2 \approx 2^{31.3}$. After filtering, we expect to remain with approximately 2^{12} valid guesses for the even bytes in a column.

The probability that we need to guess more than two odd bytes in at least one of the columns of the key is less than 2%. Thus, for over 98% of the keys, we have an effective attack.

Relationship Across Rounds. So far, we have created independent lists of even bytes for each of the halves of $\mathbf{A}_0$ and $\mathbf{A}_2$. As calculated above, for some keys we can expect these list to consist of up to 2^{12} candidates. Because these lists are independent, combining the guesses of the two halves can yield several millions of possible candidates. To further trim these lists, we exploit the known relationship between $\mathbf{A}_0$ and $\mathbf{A}_2$.

Recall that $\mathbf{A}_i = K_i \boxplus (K_\sigma \boxplus C_i)$ and that K_2 is just K_0 with the columns swapped, i.e. $K_0[0,\dots,7] = K_2[8,\dots 15]$ and $K_0[8,\dots 15] = K_2[0,\dots,7]$, hence the sum of the columns of the two is the same, i.e. $\Sigma(K_0) = \Sigma(K_2)$. Moreover, $C_0[0,\dots,7] = C_0[8,\dots,15] = \text{0x0001000100010001}$, and $C_2 = C_0 \boxplus C_0$, hence $\Sigma(C_0) = \Sigma(C_2) - \Sigma(C_0) = \text{0x0002000200020002}$. Thus,

$$\begin{aligned}\Sigma(\mathbf{A}_0) &= \Sigma(K_0 \boxplus K_\sigma \boxplus C_0) = \Sigma(K_0) + \Sigma(K_\sigma) + \Sigma(C_0)\\ &= \Sigma(K_2) + \Sigma(K_\sigma) + \Sigma(C_2) - \Sigma(C_0) = \Sigma(\mathbf{A}_2) - \Sigma(C_0) \qquad (8)\end{aligned}$$

Recall that we have four lists of guesses, one for each of $\mathbf{A}_0$, $\mathbf{A}_0$, $\mathbf{A}_2$, and $\mathbf{A}_2$. Selecting one guess from each of the lists provides us with guesses of the even bytes $\widetilde{\mathbf{A}}_0$ and $\widetilde{\mathbf{A}}_2$. We can now calculate $\widetilde{S}_0 = \widetilde{\mathbf{A}}_0 + \widetilde{\mathbf{A}}_0 + \Sigma(C_0)$

and $\widetilde{S}_2 = \widetilde{\mathbf{A}}_2$▦ $+ \widetilde{\mathbf{A}}_2$▦ where we abuse the notation to mean adding one column of the matrix to the other. By Section 4.1, if our guesses are correct, we expect the even bytes of $\widetilde{S}_0$ and $\widetilde{S}_2$ to have similar values. Specifically, we expect the corresponding least significant bytes to be identical. Because we may miss carries from odd bytes, we expect a difference of up to 1 between other corresponding even bytes. For incorrect guesses, the probability of identical bytes is 1/256 and for a difference of up to 1 between a pair of bytes is 3/256. Hence, the probability of accepting a wrong guess is $2^{-27.2}$.

We verify experimentally that after this step, for 80% of the key only one candidate survives and for 15% two candidates survive. Based on a sample of 1000 random keys, the expected number of candidates is 1.4.

4.2 Recovering Odd Bytes of $\mathbf{A_0}$

So far, we have focused on the even bytes of $\mathbf{A}_0$, showing that the expected number of candidates for those is 1.4. We now discuss guessing the odd bytes we require for the meet-in-the-middle attack.

In total, we need to guess bytes $\mathbf{A}_0$▦, which together with the even bytes will give us a guess of $\mathbf{A}_0$▦. From the side-channel oracle, we learn the two MSBs of each of those bytes, hence a naive approach would be to guess the remaining six bits of each byte, arriving at a complexity of 2^{24}.

However, using side-channel information we can reduce the number of combinations to roughly 2^{16}. Specifically, we note that on the one hand, $K_\sigma \boxplus C_0 = \mathbf{A}_0 \boxminus K_0 = \mathbf{A}_0 \boxminus K$, while on the other, $K_\sigma \boxplus C_0 = \mathbf{A}'_0 \oplus \mathbf{D}_0$. We have side-channel information on $O(\mathbf{A}) = O(K \boxplus 5) \approx O(K)$ from the generation for K_σ, and on $O(\mathbf{A}'_0)$ from the expansion of RK_0. Moreover, given a guess of $\mathbf{A}_0$▦, we can find $\mathbf{D}_0$▦. We can therefore compute two approximations of $O(K_\sigma \boxplus C_0)$, and eliminate guesses in case of a mismatch.

4.3 Recovering K_σ

To recover K_σ we first split it into four parts of four bytes each. We guess one of these parts and exploit the structure of the cipher for a meet-in-the-middle attack to derive the remaining parts.

Guess $\boldsymbol{K_\sigma[0,1,2,3]}$. We first guess K_σ▦. Note that we only need to guess the six LSBs since the two MSBs are derived from $\mathbf{D}_0 \oplus O(\mathbf{A}'_0)$, where $\mathbf{D}_0$ is known from the previous step and $O(\mathbf{A}'_0)$ is obtained from the side-channel oracle.

Derive $\boldsymbol{K_\sigma[4,5,6,7]}$. Once we know K_σ▦, we use the meet-in-the-middle attack to derive K_σ▦. As Figure 5 shows, we assume that we have a guess of $\mathbf{A}_0$▦, which determines $\mathbf{D}_0$▦. Together with a guess of K_σ▦, these determine the value of $\mathbf{A}'_0$▦ which determines $\mathbf{C}'_0$▦. From the other end, the (assumed known) value of RK_0 together with the guess of K_σ▦ allow us to determine $\mathbf{D}'_0$▦. Thus, with a guess of four bytes of a column of K_σ, we can determine four bytes in the input of the MixColumns transformation as well as four bytes in its output.

Using the MDS property of MixColumns, we can now determine the missing bytes, specifically, the value of $\mathbf{D}'_0$▥. Combining this with RK_0, allows us to determine the missing bytes of K_σ▥. We note that the information we have on the MSBs of $K_\sigma \boxplus C_0$ allows us to eliminate wrong guesses with a probability of $1 - 2^{-8}$. (Two bits for each of four bytes.)

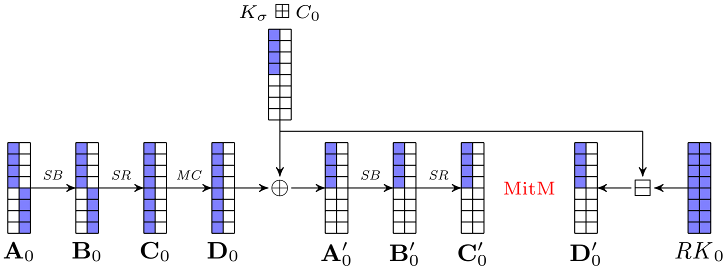


Fig. 5: Meet-in-the-middle attack recovering $K_\sigma[0, \ldots, 7]$.

Derive $\boldsymbol{K_\sigma[8, 10, 12, 14]}$. In contrast to previous step, this time we consider the relationship among $\mathbf{A}_i$, K, K_σ and C_i. The first relationship that we use is

$$\mathbf{A}_0▥ = K▥ \boxplus (K_\sigma \boxplus C_0)▥ \tag{9}$$

Recall that we know $\mathbf{A}_0$▥ and $(K_\sigma \boxplus C_0)$▥. Through subtraction, we obtain K▥ as illustrated in Figure 6a. The obtained value in byte $K[6]$ may not be exact due to a possible borrow from the unknown byte $\mathbf{A}_0[5]$. Hence, $K[6]$ could be $(\mathbf{A}_0 \boxminus (K_\sigma \boxplus C_0))[6] \pm 1$ due to carry or borrow.

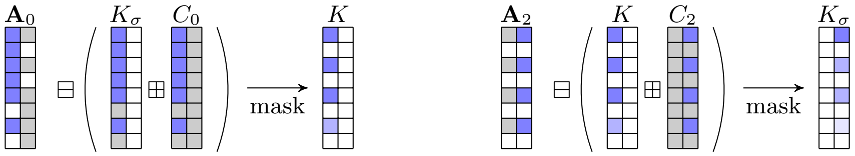
To recover the four target bytes K_σ▥, we relate the obtained K▥ to the known constant C_2 and the already known $\mathbf{A}_0$▥ through the following relationship

$$\mathbf{A}_2▥ = K▥ \boxplus (K_\sigma \boxplus C_2)▥ \tag{10}$$

where we abuse the notation to mean operations are performed as a single column on the colored columns. As illustrated in Figure 6b, K_σ▥ can be recovered by subtracting $(K$▥ $\boxplus\, C_2$▥$)$ from $\mathbf{A}_2$▥. Note that similar remark as when recovering $K[6]$ also applies here, namely, carries and/or borrows can affect the value of bytes $K_\sigma[2, 4, 6]$.

Obtain $\boldsymbol{K_\sigma[9, 11, 13, 15]}$. Since we now know half a column of K_σ▥ which also allows us to compute the corresponding half a column of $\mathbf{D}'_0$▥, we wish to apply a similar meet-in-the-middle technique as for recovering K_σ▥. To do so, another piece of information that we need is half a corresponding column of $\mathbf{C}'_0$.

Recall that we start the K_σ recovery process by guessing $(K_\sigma \boxplus C_0)$▥. This allows us to obtain $\mathbf{A}'_0$▥. Now, with the extra information of K_σ▥ obtained in



(a) Recover $K[0, 2, 4, 6]$ (b) Recover $K_\sigma[8, 10, 12, 14]$

Fig. 6: Relationships among $\mathbf{A}_i, K, K_\sigma$ and C_i to derive K_σ. The gray shade highlights bytes whose values are known but not used in this recovery. The blue shade highlights bytes whose values are known and used in this recovery. The lighter blue shade denotes bytes with uncertainly. Note that here we abuse the notation to mean performing operations as a single column of the matrix.

Section Section 4.3, we can compute $(K_\sigma \boxplus C_0)$▥ by adding the known constant C_0 which allows us to compute $\mathbf{A}'_0$▥. Then, following the sequence of SubBytes and ShiftRows provides us $\mathbf{C}'_0$▥. That is, we compute:

$$\mathbf{A}'_0▥ \leftarrow \mathbf{D}_0▥ \oplus (K_\sigma \boxplus C_0)▥$$
$$\mathbf{B}'_0▥ \leftarrow SB(\mathbf{A}'_0▥)$$
$$\mathbf{C}'_0▥ \leftarrow SR(\mathbf{B}'_0▥)$$

At this stage, we know $\mathbf{C}'_0$▥, i.e. half-column input to MixColumns and $\mathbf{D}'_0$▥, i.e. its corresponding half-column output. We can proceed with splitting the input of MixColumns and solving for the unknown input half $\mathbf{C}'_0$▥ as illustrate in Figure 7. Even though the four MixColumns output bytes that we know are not consecutive, this does not affect our technique since both $MC(\mathbf{C}'_0$▥$)$ and $MC(\mathbf{C}'_0$▥$)$ contribute to $\mathbf{D}'_0$▥. Therefore, we can simply construct a system of linear equations focusing on the four bytes $\mathbf{C}'_0$▥ and $\mathbf{D}'_0$▥ that we know and solve for the four unknown bytes $\mathbf{C}'_0$▥.

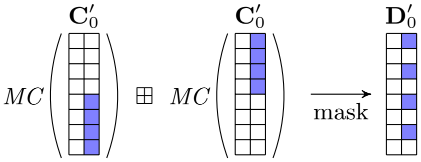


Fig. 7: Recover $\mathbf{C}'_0[8, 9, 10, 11]$

Once we know the full-column $\mathbf{C}'_0$▥, performing MixColumns allows us to recover the full-column $\mathbf{D}'_0$▥. Then, K_σ▥ can be recovered by subtracting $\mathbf{D}'_0$▥

from RK_0▥. That is, we compute:

$$\mathbf{D}'_0\text{▥} \leftarrow MC(\mathbf{C}'_0\text{▥})$$
$$K_\sigma\text{▥} \leftarrow RK_0\text{▥} \boxminus \mathbf{D}'_0\text{▥}$$

4.4 Recovering K

After we recover the entire K_σ, we can recover K by simply reversing the RK_0 expansion. For the Kalyna-l/k variants where the block size l is the same as the key length k, i.e. $l = k$, we compute the following steps:

$$\mathbf{A}'_0 \leftarrow SB^{-1}(SR^{-1}(MC^{-1}(RK_0 \boxminus (K_\sigma \boxplus C_0))))$$
$$\mathbf{A}_0 \leftarrow SB^{-1}(SR^{-1}(MC^{-1}(\mathbf{A}'_0 \oplus (K_\sigma \boxplus C_0))))$$
$$K \leftarrow \mathbf{A}_0 \boxminus (K_\sigma \boxplus C_0)$$

Once we obtain K, we can verify by computing the K_σ expansion and check against our recovered K_σ. If both match, this guarantees that we successfully recover the correct K. In other words, if our guess of either the odd bytes of $(K \boxplus (K_\sigma \boxplus C_0))$▥ or $(K_\sigma \boxplus C_0)$▥ was not correct, we would detect at this step. If that is the case, we try different candidates until we recover the correct K.

5 Attacks on Other Kalyna Variants

Most of the techniques described in Section 4 also apply to other variants of Kalyna, which have larger block size (l) and/or key length (k). This section discusses the hurdles that increasing the sizes adds and explains how we tackle them to recover the master key.

5.1 Relationship of K_i

Recall that K_i is one of the inputs to RK_i expansion. With an increase in the key length k (regardless whether the block size l also increases), the variations of K_i key part also increase. As a consequence, there are fewer K_i's that are identical to K_0.

As shown in Figure 8, with three identical K_i's, we can divide the bounds on the six LSBs of $\mathbf{A}_0$ into three groups (as indicated by the number lines). In contrast, when we have only two identical K_i's, the bounds can be divided into two groups. In a lucky case, such as in Kalyna-128/256, the size of those two groups are nearly balanced. However, the splits in other variants may not be well balanced, which results in less tight bound in a larger group. This implies an increase in the cost of guessing $\mathbf{A}_i$.

Another impact of having many variations of K_i is that it is more difficult to apply the relationship across columns. Recall that in Kalyna-128/128 there are only two variations of K_i whose difference in the column-wise sum is a known

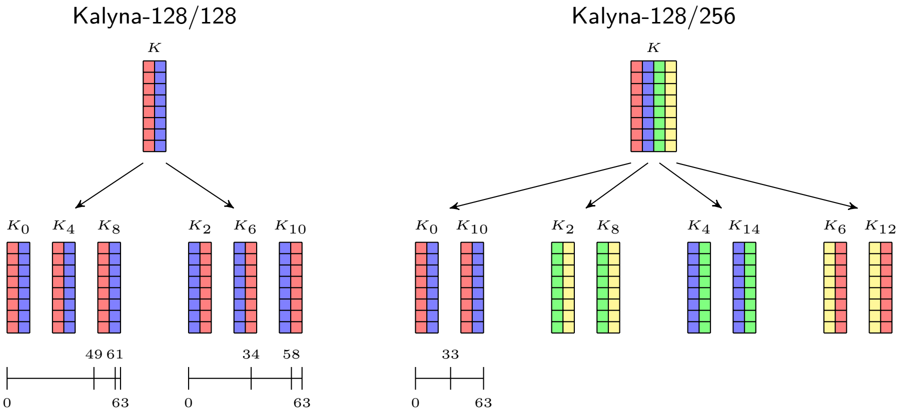


Fig. 8: The relationship between the full master key K and K_i key part used for RK_i expansion in Kalyna-128/k. The effect on the bounds of the 6 LSBs of $\mathbf{A}_i$ is presented using number lines.

constant. Therefore, the cost of applying the column-wise sum is equivalence to guessing $\mathbf{A}_i$ of those two K_i variations. For other variants of Kalyna, it requires significantly more guesses to be able to apply the column-wise sum. Take Kalyna-128/256 as an example (see Figure 8). We need to guess all four K_i variations so that the column-wise sum of, for example, K_0 and K_2 is equivalent (up to a known constant) to that of K_4 and K_6. Therefore, we opt for guessing relevant bytes of $\mathbf{A}_0$ instead of guessing $\mathbf{A}_0, \mathbf{A}_2, \mathbf{A}_4$ and $\mathbf{A}_6$.

Observe that not being able to use the relationship across column affects our attacks in two ways. First, we lose an extra filter to eliminate wrong $\mathbf{A}_i$ guesses (step 2.2 in Figure 4) thus resulting in having more candidates for subsequent steps. Second, we can no longer derive a different column of K_σ from a known column (step 3.3 in Figure 4). This forces us to guess more bits of K_σ to be able to derive the full key K.

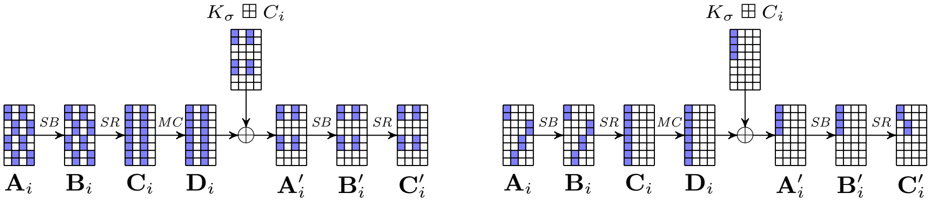
5.2 Large Constant C_i

Each RK_i expansion uses a known constant C_i which is defined as a value 0x00010001...0001 shifted by the round key index $i/2$. The length of C_i is the same as the block size l. Observe that if $0 \leq i < 16$, all even bytes of C_i are non-zero while all odd bytes of C_i are zero, which is the case for Kalyna-l/128 and Kalyna-l/256. However, in Kalyna-l/512, i can be as high as 18, which means that in rounds 16 and 18 the even bytes are zero while the odd bytes are non-zero. Therefore, we can no longer assume that the odd bytes from different round keys (but with the same K_i) are identical (step 1 in Figure 4). Thus, we need to guess odd bytes in addition to even bytes, increasing the complexity of our attacks.

5.3 Aligning Columns

One of the core techniques in our attacks is meet-in-the-middle operating columnwise (steps 3.2 and 3.4 in Figure 4). This requires us to have sufficient information aligned in the corresponding columns. The main obstacle that we face is that the round key expansion performs the ShiftRows twice, resulting in diffusion across columns. This is not a problem with Kalyna-128/k because there are only two columns where ShiftRows merely splits the columns in half. However, with Kalyna-256/k, ShiftRows spreads the information from one column into four where only one quarter (thus less than half) remains in a column (see Figure 9a). This is worse in Kalyna-512/512 where information is spread into eight columns.

To overcome the second ShiftRows, we increase the number of guessed bytes and guess the bytes in locations that would maintain their alignment. That is, instead of attempting to align a single column, we align multiple columns. We guess bytes in an alternating pattern (see Figure 9b). This pattern applies to both guessing $\mathbf{A}_i$ and K_σ.



(a) Insufficient information

(b) Sufficient and well aligned bytes

Fig. 9: Propagation and alignment of known bytes from initial guessed bytes to perform meet-in-the-middle attack in Kalyna-256/k.

6 The Practical Attack

We now proceed to empirically validate the assumptions made for the analysis. We implement a cache-based side-channel attack against the key expansion algorithm in the reference implementation of Kalyna [44] and demonstrate that we can instantiate the required oracle. We also describe how we exploit the additive key whitening used in Kalyna to efficiently recover the first round key RK_0.

6.1 Instantiating the Oracle

Experimental Setup. We implement the attack on a Dell Vostro 5581, featuring Intel Core i5-8265U CPU, with four cores and a 6 MB last-level cache, running Ubuntu-18.04.5. We use the Prime+Probe attack as implemented in the

Mastik toolkit [63]. To achieve a high temporal resolution, we use SGX-Step [56]. We note that other approaches for achieving high temporal resolution exist [4, 11, 14, 26, 31]. Thus the attack is feasible outside SGX.

Victim. The victim is an SGX enclave that runs the reference implementation of Kalyna [44]. The implementation uses four S-Boxes, each 256-bytes long, occupying four consecutive cache lines each, to a total of 16 consecutive cache lines. The cache attack identifies the cache line accessed. Hence, with four cache lines per S-Box, the attack recovers the two most significant bits of the index.

Attack. As in past works, before the attack we disable frequency scaling, automatic power management, and Intel Turbo Boost. We further isolate the core that runs the victim and disable the cache prefetcher. We use a controlled-channel attack [61] to stop enclave execution at the start of key expansion. That is, we mark the page containing the S-Boxes as not existing, forcing an interrupt when the enclave executes the first SubBytes operation. We then use SGX-Step to single-step the enclave. We use Mastik to prime the cache sets that hold the S-Boxes and mark the page that contains the S-Boxes as not-accessed prior to each single step. After stepping, if the page has been access, we probe the cache sets of the S-Boxes and record the results. We stop after 240 S-Box accesses.

Results. Similar to prior works [26, 31, 42], we find that even though we only execute a single instruction, we can observe multiple accesses to the S-Boxes. After filtering the accesses, we can correctly identify 96.55% of the memory accesses, with the only failures being on Byte 7 of the state. In cases of failures, we have two or three options for the oracle of Byte 7, depending on the subsequent accesses to S-Box 3.

Summary. The attack takes several seconds and retrieves S-Box accesses with a high success rate. Thus, it can be used as the oracle for our key-recovery attack.

6.2 Recovering the First Round Key

In Section 4.3 we use RK_0 as part of recovering the master key. We now show how we can exploit the additive key whitening of Kalyna to easily recover RK_0.

In AES and in many similar ciphers, the first operation during encryption is to apply Boolean whitening to the plaintext, i.e. to XOR it with a key. In contrast, Kalyna uses additive whitening. Specifically, it computes $P \boxplus RK_0$ where P denotes a plaintext, and uses the result as the input to the first round. Because (arithmetic) addition is not a linear operation in $GF(2)$, additive key whitening hinders cryptanalysis [35, 43].

We observe, however, the additive masking is inferior when it comes to protection against cache attacks. The reason is that cache attacks observe specific bits of the data. For example, the oracle we use observes the two MSBs of the input to an S-Box. When Boolean whitening is used, changes in the plaintext remain local, i.e. a change in the plaintext only affects the changed bit but not other bits and cannot, therefore, expose more information to the attacker.

In contrast, with additive whitening, a change of a bit can change the pattern of ripples of the carries, resulting in changes in other bits, which can reveal more information. To exploit this property, we first encrypt an all-zero plaintext and use our attack to recover the oracle for the SubBytes operation of the first round. Because $P = 0$, we have $P \boxplus RK_0 = RK_0$, and the oracle reveals the two MSBs of each byte of RK_0.

We now encrypt a plaintext where all the bytes have the value 32. We note that if bit five of a key byte is zero, adding 32 will not change the values of bits six or seven of the sum. Consequently, the oracle reading will be the same as for the all-zero plaintext. If, however, bit five of the key byte is set, adding 32 will cause a carry to bit six and the oracle reading will be different. Thus with the value 32 we learn the value of bit five of each byte in RK_0.

In the next step we repeat the process, this time with a value of 16 for bytes where bit five of the key is set and a value of 48 where it is clear. Using the same argument, we now learn bit four of the key. We note that this is, basically, a binary search that exploits carry ripples to recover the next bit. Hence, using only seven adaptively chosen plaintexts, we can completely recover RK_0.

6.3 Recover the Master Key K of Kalyna-128/128

Experimental Setup. To validate our cryptanalysis, we performed a practical attack on recovering the master key K of Kalyna-128/128 on a compute cluster. We spawn a task for each 16 guesses of the odd bytes of $\mathbf{A}_0$. During the experiment, we record the task execution time and the execution time of testing a single candidate.

Practical Attack. To prove that the attack is practical, we picked a random secret key ensuring no overflows in $K_\sigma \boxplus C_i$ when recovering the K_σ. We note that overflows facilitate the attack, hence this choice represent a conservative estimate. For the selected key, there are 69,249 candidates for odd bytes of $\mathbf{A}_0$. We spawn 4,329 tasks, each of which tests 16 candidates. (The last task only tests one candidate.) We then let the cluster schedule the tasks according to its scheduling policies.

Results. To determine the worst-case execution time we run all 4,329 tasks to completion. The key was found after 37 hours, but full completion took 49 hours. Testing a wrong guess takes on average about 44 minutes. The full attack takes about 50 K CPU hours.

Acknowledgements

We would like to thank all reviewers for the insightful feedback, which has improved the paper.

This work was supported by the ARC Discovery Early Career Researcher Award (project number DE200101577); the ARC Discovery Project (project number DP210102670); the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; The Blavatnik ICRC at Tel-Aviv University; the National Science Foundation under grant CNS-1954712; the Phoenix HPC service at the University of Adelaide; and gifts from AMD, Google, and Intel.

References

1. Acıiçmez, O.: Yet another microarchitectural attack: exploiting I-cache. In: CSAW (2007)
2. Acıiçmez, O., Koç, Ç.K., Seifert, J.: Predicting secret keys via branch prediction. In: CT-RSA (2007)
3. Akshima, Chang, D., Ghosh, M., Goel, A., Sanadhya, S.K.: Single key recovery attacks on 9-round Kalyna-128/256 and Kalyna-256/512. In: ICISC (2015)
4. Allan, T., Brumley, B.B., Falkner, K.E., van de Pol, J., Yarom, Y.: Amplifying side channels through performance degradation. In: ACSAC (2016)
5. AlTawy, R., Abdelkhalek, A., Youssef, A.M.: A meet-in-the-middle attack on reduced-round Kalyna-$b/2b$. IEICE Trans. Inf. Syst. **99-D**(4) (2016)
6. Belarus Standard STB 34.101.31-2011: Information technology and security data encryption and integrity algorithms. Available at http://apmi.bsu.by/assets/files/std/belt-spec27.pdf (2011)
7. Bernstein, D.J., Breitner, J., Genkin, D., Groot Bruinderink, L., Heninger, N., Lange, T., van Vredendaal, C., Yarom, Y.: Sliding right into disaster: Left-to-right sliding windows leak. In: CHES (2017)
8. Bonneau, J., Mironov, I.: Cache-collision timing attacks against AES. In: CHES (2006)
9. Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.: Software grand exposure: SGX cache attacks are practical. In: WOOT (2017)
10. Bulck, J.V., Piessens, F., Strackx, R.: Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In: CCS (2018)
11. Cabrera Aldaya, A., Brumley, B.B.: HyperDegrade: From GHz to MHz effective CPU frequencies. arXiv 2101.01077
12. Cabrera Aldaya, A., Brumley, B.B., ul Hassan, S., Pereida García, C., Tuveri, N.: Port contention for fun and profit. In: IEEE SP (2019)
13. Cabrera Aldaya, A., García, C.P., Tapia, L.M.A., Brumley, B.B.: Cache-timing attacks on RSA key generation. TCHES **2019**(4) (2019)
14. Chakraborty, A., Bhattacharya, S., Alam, M., Patranabis, S., Mukhopadhyay, D.: RASSLE: return address stack based side-channel leakage. TCHES **2021**(2) (2021)
15. Chuengsatiansup, C., Feutrill, A., Sim, R.Q., Yarom, Y.: RSA key recovery from digit equivalence information. In: ACNS (2022)
16. Compton, K.J., Timm, B., VanLaven, J.: A simple power analysis attack on the Serpent key schedule. ePrint Archive 2009/473 (2009)
17. Dall, F., De Micheli, G., Eisenbarth, T., Genkin, D., Heninger, N., Moghimi, A., Yarom, Y.: CacheQuote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. TCHES **2018**(2) (2018)
18. Dassance, F., Venelli, A.: Combined fault and side-channel attacks on the AES key schedule. In: FDTC (2012)
19. Duman, O., Youssef, A.M.: Fault analysis on Kalyna. Inf. Secur. J. A Glob. Perspect. **26**(5) (2017)
20. Fernandes Medeiros, S., Gérard, F., Veshchikov, N., Lerman, L., Markowitch, O.: Breaking Kalyna 128/128 with power attacks. In: SPACE (2016)
21. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. JCEN **8**(1) (2018)
22. Genkin, D., Pachmanov, L., Tromer, E., Yarom, Y.: Drive-by key-extraction cache attacks from portable code. In: ACNS (2018)

23. Genkin, D., Poussier, R., Sim, R.Q., Yarom, Y., Zhao, Y.: Cache vs. key-dependency: Side channeling an implementation of Pilsung. TCHES **2020**(1) (2020)
24. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the line: Practical cache attacks on the MMU. In: NDSS (2017)
25. Gruss, D., Spreitzer, R., Mangard, S.: Cache template attacks: Automating attacks on inclusive last-level caches. In: USENIX Security (2015)
26. Gullasch, D., Bangerter, E., Krenn, S.: Cache games - bringing access-based cache attacks on AES to practice. In: IEEE SP (2011)
27. Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space ASLR. In: IEEE SP (2013)
28. Irazoqui, G., Eisenbarth, T., Sunar, B.: Systematic reverse engineering of cache slice selection in Intel processors. In: DSD (2015)
29. Irazoqui Apecechea, G., Eisenbarth, T., Sunar, B.: S$A: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In: IEEE SP (2015)
30. Irazoqui Apecechea, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a minute! A fast, cross-VM attack on AES. In: RAID (2014)
31. Kayaalp, M., Abu-Ghazaleh, N.B., Ponomarev, D.V., Jaleel, A.: A high-resolution side-channel attack on last-level cache. In: DAC (2016)
32. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: CRYPTO (1996)
33. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: CRYPTO (1999)
34. Kryptos Logic: A brief look at North Korean cryptography. https://www.kryptoslogic.com/blog/2018/07/a-brief-look-at-north-korean-cryptography/ (Jul 2018)
35. Kumar Gupta, S., Ghosh, M., Mohanty, S.K.: Cryptanalysis of Kalyna block cipher using impossible differential technique. In: ICMC (2020)
36. Lin, L., Wu, W.: Improved meet-in-the-middle attacks on reduced-round Kalyna-128/256 and Kalyna-256/512. DCC **86**(4) (2018)
37. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: IEEE SP (2015)
38. MacWilliams, F.J., Sloane, N.: The Theory of Error-Correcting Codes. North-Holland Publishing Company (1977)
39. Mangard, S.: A simple power-analysis (SPA) attack on implementations of the AES key expansion. In: ICISC (2002)
40. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks - revealing the secrets of smart cards. Springer (2007)
41. Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., Francillon, A.: Reverse engineering Intel last-level cache complex addressing using performance counters. In: RAID (2015)
42. Moghimi, A., Irazoqui, G., Eisenbarth, T.: CacheZoom: How SGX amplifies the power of cache attacks. In: CHES (2017)
43. Mukhopadhyay, D., Chowdhury, D.R.: Key mixing in block ciphers through addition modulo 2^n. ePrint Archive 2005/383 (2005)
44. Oliynykov, R.: Kalyna block cipher reference implementation. https://github.com/Roman-Oliynykov/Kalyna-reference (2015), visited: 6 December 2021
45. Oliynykov, R., Gorbenko, I., Kazymyrov, O., Ruzhentsev, V., Kuznetsov, O., Gorbenko, Y., Dyrda, O., Dolgov, V., Pushkaryov, A., Mordvinov, R., Kaidalov, D.: A new encryption standard of Ukraine: The Kalyna block cipher. ePrint Archive 2015/650 (2015)

46. Ortiz, J.J.G., Compton, K.J.: A simple power analysis attack on the twofish key schedule. CoRR **abs/1611.07109** (2016)
47. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: CT-RSA (2006)
48. Percival, C.: Cache missing for fun and profit. In: Proceedings of BSDCan (2005), https://www.daemonology.net/papers/htt.pdf
49. Pereida García, C., Brumley, B.B.: Constant-time callees with variable-time callers. In: USENIX Security (2017)
50. Pereida García, C., Brumley, B.B., Yarom, Y.: "Make sure DSA signing exponentiations really are constant-time". In: CCS (2016)
51. Pessl, P., Groot Bruinderink, L., Yarom, Y.: To BLISS-B or not to be: Attacking strongSwan's implementation of post-quantum signatures. In: CCS (2017)
52. Schneier, B., Kelsey, J., Whiting, D., Ferguson, N., Wagner, D., Hall, C.: Twofish: A 128-bit block cipher. In: First AES Conference (1998)
53. Schneier, B., Kelsey, J., Whiting, D., Wagner, D.A., Hall, C.: On the Twofish key schedule. In: SAC (1998)
54. Shishkin, V., Dygin, D., Lavrikov, I., Marshalko, G., Rudskoy, V., Trifonov, D.: Low-weight and hi-end: Draft Russian encryption standard. In: Current Trends in Cryptology (CTCrypt) (2014)
55. Shusterman, A., Kang, L., Haskal, Y., Meltser, Y., Mittal, P., Oren, Y., Yarom, Y.: Robust website fingerprinting through the cache occupancy channel. In: USENIX Security (2019)
56. Van Bulck, J., Piessens, F., Strackx, R.: SGX-Step: A practical attack framework for precise enclave execution control. In: SysTex (2017)
57. Wagner, M., Heyse, S.: Single-trace template attack on the DES round keys of a recent smart card. ePrint Archive 2017/57 (2017)
58. Wagner, M., Heyse, S.: Improved brute-force search strategies for single-trace and few-traces template attacks on the DES round keys. ePrint Archive 2018/937 (2018)
59. Wang, G., Zhu, C.: Single key recovery attacks on reduced AES-192 and Kalyna-128/256. Sci. China Inf. Sci. **60**(9) (2017)
60. Wichelmann, J., Moghimi, A., Eisenbarth, T., Sunar, B.: MicroWalk: A framework for finding side channels in binaries. In: ACSAC (2018)
61. Xu, Y., Cui, W., Peinado, M.: Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In: IEEE SP (2015)
62. Yan, M., Fletcher, C.W., Torrellas, J.: Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In: USENIX Security (2020)
63. Yarom, Y.: Mastik: A micro-architectural side-channel toolkit. https://cs.adelaide.edu.au/~yval/Mastik (2016)
64. Yarom, Y., Falkner, K.: Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In: USENIX Security (2014)
65. Yarom, Y., Ge, Q., Liu, F., Lee, R.B., Heiser, G.: Mapping the Intel last-level cache. ePrint Archive 2015/905 (2015)
66. Yarom, Y., Genkin, D., Heninger, N.: CacheBleed: a timing attack on OpenSSL constant-time RSA. JCEN **7**(2) (2017)
67. Yuce, B., Schaumont, P., Witteman, M.: Fault attacks on secure embedded software: Threats, design, and evaluation. J. Hardw. Syst. Secur. **2**(2) (2018)