Bricklayer Attack: A Side-Channel Analysis on the ChaCha Quarter Round

Alexandre Adomnicai[1,3], Jacques J.A. Fournier[2], and Laurent Masson[1]

[1] Trusted Objects, Aix-en-Provence, France,
{a.adomnicai, l.masson}@trusted-objects.com
[2] CEA-Leti, Grenoble, France,
jacques.fournier@cea.fr
[3] EMSE, Gardanne, France,

Abstract. ChaCha is a family of stream ciphers that are very efficient on constrainted platforms. In this paper, we present electromagnetic side-channel analyses for two different software implementations of ChaCha20 on a 32-bit architecture: one compiled and another one directly written in assembly. On the device under test, practical experiments show that they have different levels of resistance to side-channel attacks. For the most leakage-resilient implementation, an analysis of the *whole* quarter round is required. To overcome this complication, we introduce an optimized attack based on a divide-and-conquer strategy named *bricklayer attack*.

Keywords: ChaCha, Implementation, Side-channel attacks

1 Introduction

ChaCha [7] is a family of stream ciphers introduced by Daniel J. Bernstein in 2008. It is a variant of the Salsa20 family [8], which is part of the eSTREAM portfolio [4], providing better diffusion for similar performances. ChaCha is an ARX-based cipher, which means that it only uses modular additions, rotations and bitwise XORs. It has been widely adopted for encryption, as well as for random number generation in many operating systems (*e.g.* Linux, OpenBSD) and protocols (*e.g.* SSH, TLS). Moreover, the upcoming version 1.3 of the Transport Layer Security (TLS) protocol [36] will allow Authenticated Encryption with Associated Data (AEAD) cipher suites only, leaving AES-CCM [32], AES-GCM [38] and ChaCha20-Poly1305 [26] as the only three options. This update should significantly increase the use of ChaCha in the near future. On top of that, the Internet of Things (IoT) should be in favour of the ChaCha deployment (*e.g.* Apple HomeKit for IoT devices [2]), since its instances are cheaper than AES on microcontrollers that do not have any dedicated cryptographic hardware. For instance, on Android phones, HTTPS connections from Chrome browsers to Google now use ChaCha20-Poly1305 [12].

As a result of its standardization, ChaCha is under close scrutiny with regards to cryptanalysis, especially regarding differential attacks [3,42,39,14,29].

Recently, studies have been carried out to evaluate its physical security, especially regarding fault attacks [33,25]. However, only one side-channel analysis has been proposed so far [22]. We believe that further work must be undertaken in this field since ChaCha is particularly well suited for embedded devices.

Our contribution. In this paper, we focus on the side-channel analysis of ChaCha by taking two different implementations into consideration.

First, we investigate the OpenSSL C source code compiled on a 32-bit ARM microcontroller. It results in a straightforward attack path, which consists in targeting each 32-bit key word independently.

The second target is an assembly implementation which saves some memory accesses. We highlight that, on the device under test (DUT), this slight modification protects from the only side-channel attack published to date. Nevertheless, our implementation remains vulnerable even though attack paths are more complex. We tackle this problem by introducing the *bricklayer attack*, which is based on a divide-and-conquer approach, and emphasize that attacking from the keystream rather than from the input is way more efficient.

Outline. First, we present the ChaCha family of stream ciphers before providing an outline of side-channel attacks. Then, we describe our approaches on performing electromagnetic analyses depending on software implementations of ChaCha. Subsequently, we present our practical results and discuss the feasibility of conducting these attacks in real-world scenarios. Finally, we analyze the overhead introduced by the masking countermeasure in the specific case of ChaCha20.

2 The ChaCha family of stream cipher

As its predecessor, and unlike traditional stream ciphers, ChaCha does not have an initilization phase since it works like a block cipher used in counter (CTR) mode [19]. Its core is an ARX-based function which maps a 512-bit input block to a 512-bit output key stream. Input blocks are built by arranging data in a 4×4 matrix where each element is a 32-bit word. The encryption key fills half of the matrix as it is 256-bit long, while the two remaining quarters are repectively occupied by the inputs and the constant '`expand 32-byte k`'. This constant aims at reducing the amount of data an attacker can control while the inputs refer to a nonce which is built from the block counter and the initial vector (IV).

`'expa'`	`'nd 3'`	`'2-by'`	`'te k'`
k_0	k_1	k_2	k_3
k_4	k_5	k_6	k_7
$\texttt{nonce}_0$	$\texttt{nonce}_1$	$\texttt{nonce}_2$	$\texttt{nonce}_3$

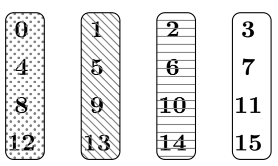
Fig. 1: ChaCha's input block intialization

The core function is defined by iterating several rounds on the input block, where each round consists of four parallel quarter round (QR) operations. A QR updates 4 words (*i.e.* a block quarter) as defined in Algorithm 1 where $\boxplus$ means addition modulo 2^{32}, $\oplus$ means XOR and $\lll$ means left bitwise rotation.

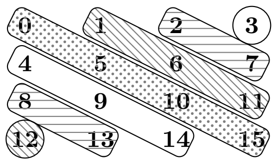
Algorithm 1 ChaCha `quarterround(a, b, c, d)`

```
a ⊞= b;        d ⊕= a;        d ⋘= 16;
c ⊞= d;        b ⊕= c;        b ⋘= 12;
a ⊞= b;        d ⊕= a;        d ⋘= 8;
c ⊞= d;        b ⊕= c;        b ⋘= 7;
```

Depending on the round number (enumerated from 0), each QR operates either on a column, or on a diagonal. ChaCha$R$ refers to a specific instance



(a) Even round



(b) Odd round

where R rounds are used. Several variants are defined with 8, 12 or 20 rounds, defining different trade-offs between security and performance. Recently, it has been shown under certain assumptions that ChaCha12 is sufficiently secure to ensure a 256-bit security level [14]. Nevertheless, ChaCha20 remains the most widespread instance for security margins. In many implementations, ChaCha$R$ uses $\frac{R}{2}$ iterations of double rounds instead of R rounds, which consists in a column round and a diagonal one.

On top of iterating several rounds on the input block, an additional step is required. The reason is that while QRs scramble blocks beyond recognition, they are invertible. Therefore, applying the reverse of each operation in the reverse order leads to the original block and thus, the encryption key. ChaCha prevents this by adding the original block to the scrambled one, word by word, in order to generate the pseudo-random block. The whole encryption process is detailed in Algorithm 2.

3 Background on Side-Channel Attacks

3.1 Correlation Electromagnetic Analysis

Cryptographic primitives are usually built to resist to mathematical cryptanalysis or exhaustive key search. However, they are designed to be finally executed on a given processor with its own physical characteristics. Electronic circuits are

Algorithm 2 ChaCha$R$ encryption

```
Require:
  $n$-bit plaintext $P$
  encryption key $k$
  counter $ctr$
  IV $iv$
Ensure: $n$-bit ciphertext $C$
  for $i$ from 0 to $\lfloor \frac{n}{512} \rfloor$ do
    $B \leftarrow$ init($k, ctr, iv$)                      ▷ input block initialization
    $B' \leftarrow B$                                      ▷ working variable
    for $j$ from 0 to $\frac{R}{2} - 1$ do
      quarterround($B'_0, B'_4, B'_8, B'_{12}$)            ▷ column rounds
      quarterround($B'_1, B'_5, B'_9, B'_{13}$)
      quarterround($B'_2, B'_6, B'_{10}, B'_{14}$)
      quarterround($B'_3, B'_7, B'_{11}, B'_{15}$)
      quarterround($B'_0, B'_5, B'_{10}, B'_{15}$)         ▷ diagonal rounds
      quarterround($B'_1, B'_6, B'_{11}, B'_{12}$)
      quarterround($B'_2, B'_7, B'_8, B'_{13}$)
      quarterround($B'_3, B'_4, B'_9, B'_{14}$)
    end for
    $B \leftarrow B \boxplus B'$                           ▷ final block addition
    $C_i \leftarrow P_i \oplus B$
    $ctr \leftarrow ctr + 1$
  end for
```

inherently leaky as they produce emissions that make it possible for an attacker to deduce how the circuit works and what data is being processed. Because these emissions are nothing more than side effects, their use to recover cryptographic keys has been termed 'side-channel attacks'. Since the publication of Differential Power Analysis (DPA) [24], it is common knowledge that the analysis of the power consumed by the execution of a cryptographic primitive might reveal information about the secret involved.

A few years later, Correlation Power Analysis (CPA) has been widely adopted over DPA as it requires fewer traces and has been shown to be more efficient [11]. The principle is to target a sensitive intermediate state of the algorithm and try to predict its value from the known input and different key guesses. Then, to uncover the link between these predictions and the leakage measurements, the Pearson correlation coefficient between these two variables is computed using an appropriate leakage model. The Hamming weight (HW) and the Hamming distance (HD) model are the most commonly used models to simulate the leakage of a cryptographic device. For each key hypothesis, it results in a value between -1 (total negative correlation) and 1 (total positive correlation) for every point in time, indicating how much the prediction correlates with the recorded values over several measurements. The formula of this coefficient is

$$\mathsf{Corr}(X,\ Y) = \frac{\mathsf{E}(X \cdot Y) - \mathsf{E}(X) \cdot \mathsf{E}(Y)}{\sqrt{\mathsf{E}\left((X - \mathsf{E}(X))^2\right) \cdot \mathsf{E}\left((Y - \mathsf{E}(Y))^2\right)}} \qquad (1)$$

where $\mathsf{E}(X)$ is the expected value of the random variable X. Finally, the hypothesis which matches with the real key should return a significantly higher coefficient than the other hypotheses. This attack remains valid when analyzing electromagnetic emanations [20,35] instead of power consumption, since they are mainly due to the displacement of current through the rails of the metal layers. In this case, we talk about Correlation ElectroMagnetic Analysis (CEMA).

3.2 Selection Function

The intermediate state y on which the side-channel attack focuses is defined by a *selection function* $\varphi(x, k) = y$, which is part of the encryption algorithm. It depends on x, a known part of the input and on k, an unknown part of the secret key. Usually, selection functions are chosen to be easy to compute, typically at the beginning of the encryption or decryption process. Furthermore, a valuable property for selection functions is high non-linearity as it ensures a good distinguishability between the correct and incorrect key guesses. Indeed, correlation between the leakage and the prediction will be close to zero if the key guess is incorrect due to their non-linear relationship.

In case of ARX structures, the non-linearity only relies on modular additions, while diffusion is provided by rotations (diffusion within single words) and XORs (diffusion between words). Although the carry propagation in the modular addition results in some non-linearity, it is not as good a candidate as S-boxes. It can be explained by the fact that most significant bits in the output of a modular addition are more subject to non-linearity than least significant ones. However, side channel attacks remain possible as shown in numerous publications [27,10,43].

4 Side-Channel Overview of ChaCha

4.1 ChaCha Case Study

To set up such a side-channel attack, one has to determine an attack path (*i.e.* to choose a selection function) either starting from the plaintext, or from the ciphertext. Physical attacks against stream ciphers can be challenging because the key stream is computed independently from the plaintext/ciphertext, which interferes in the relationship between known values and the secret key. However, from a side-channel point of view, ChaCha differs significantly from other stream ciphers' designs such as linear-feedback shift registers where the key is only directly involved during registers' initialization. Indeed, as ChaCha operates like a block cipher in CTR mode, the key is directly manipulated everytime a 512-bit block needs to be encrypted. More precisely, each key word directly interacts with other data during the first round (after which they have been updated) and again during the final block addition.

An attack that takes advantage of the first round has already been published in [22]. The attack on the i^{th} column round $(0 < i < 4)$ relies on the selection function defined by

$$\varphi_0 \left(\texttt{nonce}_i, \tilde{k}_i \parallel k_{i+4} \right) = \left(\left(\texttt{nonce}_i \oplus \tilde{k}_i \right) \lll 16 \right) \boxplus k_{i+4} \tag{2}$$

where $\tilde{k}_i = k_i \boxplus \texttt{constant}_i$. Hence, as soon as a couple $(\tilde{k}_i, k_{i+4})$ has been recovered, one just has to subtract the corresponding constant word in order to get k_i. However, this selection function forces the attacker to target two key words at once, which results in a key search space $|\mathcal{K}| = 2^{64}$. Since the bit-size of the targeted subkey determines the memory complexity of the side-channel attack, one can understand why this would be undoable in practice. To get around this problem, the authors exploit the QRs' intermediate states in order to operate step by step. They propose to first recover k_i by targeting $\texttt{nonce}_i \oplus \tilde{k}_i$ and then take advantage of its knowledge to find k_{i+4}. Therefore, recovering k_i and k_{i+4} requires the knowledge of $\texttt{nonce}_i$. However, the paper also describes an attack path that allows to recover the entire key with the knowledge of only two words. This latter exploits several intermediate states in the first two rounds.

Regarding the final block addition, an attacker could choose $\varphi(x, k) = x \boxminus k$ where x refers to a keystream word and $\boxminus$ refers to modular subtraction. Compared to the previous attack path, it has the advantage of recovering all key words using the modular substraction as selection function. Moreover, all keystream words are pseudorandom values, which is not necessarily the case for nonces. However, this selection function requires the knowledge of the keystream (*i.e.* both plaintext and ciphertext).

Throughout this paper, we will make the assumption that an attacker has access to all this information. In Sect. 6 we discuss the attacks' feasibility in practice and thus, whether our assumptions are reasonable.

4.2 Implementation Aspects

When targeting software implementations on load/store architectures, data transfers due to memory accesses (*i.e.* loads and stores between memory and registers) are known to leak the most information compared to arithmetic and logic operations [31,13], which only occur between registers and are usually unexploitable in practice [9]. Our practical experiments on the DUT presented in Sect. 6 verified this hypothesis. Therefore, the intermediate values that are manipulated by these senstive operations should be easiest to target, introducing a direct link between selection functions and implementation aspects.

Throughout this paper we will study selection functions in relation to memory accesses, assuming they are the main source of exploitable leakage.

4.3 OpenSSL Implementation

First, we decided to attack a C implementation of ChaCha20 in order to see how compilers can deal with ARX structures and memory accesses. To do so, we compiled the ChaCha20 C implementation from OpenSSL (version `1.0.1f`) for an ARM Cortex-M3 microcontroller using the GNU ARM C compiler 5.06 (update 2). Regardless of the optimization level chosen (from `-O0` to `-O3`), as detailed in Appendix A, within a QR, each addition and each rotation is followed by a `STR` instruction. Hence, these memory accesses allowed us to carry out the attacks described above. Practical results are briefly presented in Sect. 6 for comparative purposes.

4.4 Side-Channel Analysis of the Salsa20 Quarter Round

In the next section, we show how memory accesses can be easily managed to remove the leakage of intermediate states within a QR. This implies to target the QR output without taking its intermediate values into consideration, making the attacks presented in [22] irrelevant in this case. Although such an analysis has already been performed on Salsa20 [30], it does not apply to ChaCha.

Algorithm 3 Salsa20 `quarterround(a, b, c, d)`

```
b ⊕= (a ⊞ d) ⋘ 7;        c ⊕= (b ⊞ a) ⋘ 9;
d ⊕= (c ⊞ b) ⋘ 13;       a ⊕= (d ⊞ c) ⋘ 18;
```

In the case of Salsa20, as described in Algorithm 3, the update of the second input only depends on itself and two others (the first and the last). This allows to recover the key words involved in this computation as first/last input words, with two other 'non-key' operands (*i.e.* constant and nonce). The attack consists in performing a CPA on a 32-bit value using a divide-and-conquer (D&C) approach, which consists in separating the attack into $\lceil \frac{32}{n} \rceil$ computations on n-bit windows in parallel. The other key words that do not match these requirements were retrieved by using the knowledge of those which have been previously recovered. This allowed to keep a search space of 2^{32} instead of 2^{64}. On top of providing better diffusion, the ChaCha QR gives *each* input word a chance to affect the other three twice. This adjustment makes the attack irrelevant against ChaCha since the key search space cannot be less than 2^{64} in any case.

5 Side-Channel Analysis of the Quarter Round

Throughout this section, for greater clarity, we assume that all operators are left-associative so that

$$a \boxplus b \oplus c \lll d \Longleftrightarrow (((a \boxplus b) \oplus c) \lll d)\,.$$

5.1 Optimizing Memory Accesses

A solution to overcome attacks on intermediate states within QRs is a straightforward assembly implementation, which is a good way to reduce memory access instructions for load/store architectures. As explained in [9], for some instances of ARX lightweight block ciphers like Simon and Speck [5], it is possible to keep the whole state in registers during the entire encryption process. Thereby, they can be implemented in assembly without having to execute a single `STR` instruction during the whole encryption process, drastically reducing the amount of leakage.

Unfortunately, in the case of ChaCha, the state consists of 16 32-bit words. Therefore, it would require a 32-bit CPU with at least 16 general-purpose registfers (excluding the stack pointer, the program counter and other specific cases such as hardwired registers) to avoid memory accesses. As our chip only has 13 general-purpose registers, we implemented ChaCha so that word values are loaded into registers at the beginning of each QR and are then stored in RAM at the end. Furthermore, during the last round, related key words are also loaded into registers at the beginning of QRs, resulting in

$$\begin{aligned}
&\texttt{quarterround}'(x_0, x_5, x_{10}, x_{15}, k_1, k_6)\\
&\texttt{quarterround}'(x_1, x_6, x_{11}, x_{12}, k_2, k_7)\\
&\texttt{quarterround}'(x_2, x_7, x_8, x_{13}, k_3, k_4)\\
&\texttt{quarterround}'(x_3, x_4, x_9, x_{14}, k_0, k_5)
\end{aligned}$$

where $\texttt{quarterround'}(a,b,c,d,x,y) = \texttt{quarterround}(a,b,c,d) \boxplus (0,x,y,0)$. This method protects against leakages that would allow an attack from the keystream using the modular subtraction as selection function. Thus, these elementary implementation tricks imply to analyze the side-channel resilience of the *whole* QR.

5.2 Focusing on the Quarter Round

As every word influences the three others, and is updated twice, the simplest selection function would be defined by focusing, during the first column rounds, on the word which is completely updated at first, resulting in having

$$\varphi_1\left(\texttt{nonce}_i, k_i \parallel k_{i+4}\right) = \texttt{nonce}_i \oplus \tilde{k}_i \lll 16 \boxplus k_{i+4} \oplus k_i \lll 12 \boxplus \tilde{k}_i\,. \tag{3}$$

However, as previously mentioned, this implies a side-channel attack on 64 bits, which is not feasible in practice. Therefore, we investigated the relevance of the D&C approach in this specific case. Figure 3 sketches how key words are involved in computations. It results that targeting n bits of $y = \varphi_1(\texttt{nonce}_i, k_i \parallel k_{i+4})$ does not lead to a complexity equal to 2^{2n} since rotations make different n-bit windows interact with each other. As there is a rotation of 16 bits followed by another one of 12, some bits of $\tilde{k}_i$ may overlap. Hence, the key search space depends on the windows' size.

$$|\mathcal{K}| = \begin{cases} 2^{4n}, & \text{if } n \leq 4\\ 2^{3n+4}, & \text{if } 4 \leq n \leq 12\\ 2^{2n+16}, & \text{if } 13 \leq n \leq 16\\ 2^{n+32}, & \text{otherwise} \end{cases} \tag{4}$$

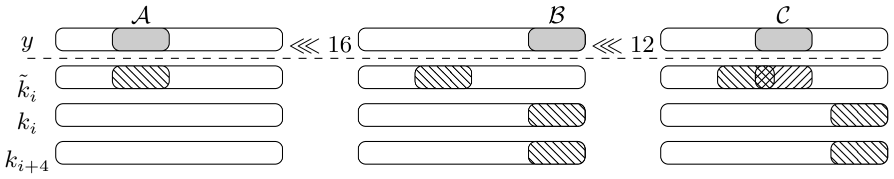


Fig. 3: D&C approach on the ChaCha QR, $n = 8$

Furthermore, rotations are discarded from the selection function, resulting in

$$\varphi_{2,n}\left(\texttt{nonce}_i, \tilde{k}_i^{\mathcal{A}} \parallel k_i^{\mathcal{B}} \parallel k_{i+4}^{\mathcal{B}} \parallel \tilde{k}_i^{\mathcal{C}}\right) = \texttt{nonce}_i^{\mathcal{A}} \oplus \tilde{k}_i^{\mathcal{A}} \boxplus_n k_{i+4}^{\mathcal{B}} \oplus k_i^{\mathcal{B}} \boxplus_n \tilde{k}_i^{\mathcal{C}} \qquad (5)$$

where superscripts refer to intervals that define n-bit windows.

In order to evaluate this method, we performed software simulations using the HW model (without any additional noise) and random nonces. As expected, the right key matches with the highest correlation coefficient. Nevertheless, some other hypotheses also lead to the maximum coefficient as shown in Fig. 4, resulting in collisions.

Definition 1 (Collision). *Let $\varphi(n,k)$ be a selection function and κ be the right key hypothesis. A collision is an hypothesis κ' such that $\varphi(n,\kappa) = \varphi(n,\kappa')$ for all n.*

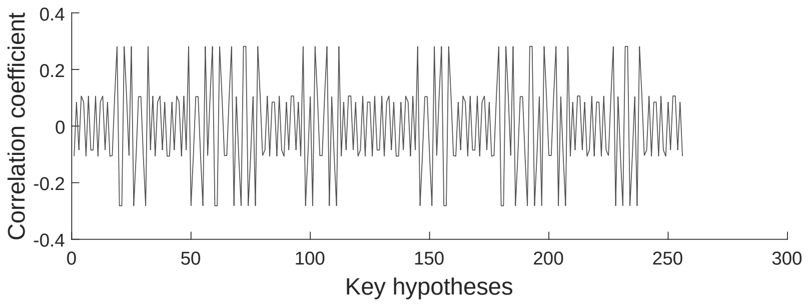


Fig. 4: Attack simulation on $\varphi_{2,2}$

This observation bring us to Proposition 1 whose proof is given in Appendix B.

Proposition 1. *An attack on $\varphi_{2,n}$ returns up to $n \cdot 2^{n+2}$ collisions.*

Another point that has not been discussed so far is the drawback caused by carry propagations. Except when focusing on the least significant bits (LSBs), one has no way of knowing if subkeys involved in additions are affected by a carry. Thus, the positions of targeted windows are very important. Plus, we made the choice to dissociate $\tilde{k}_i$ from k_i in order to prevent from erroneous predictions of $k_i^{\mathcal{A}} \boxplus_n \texttt{constant}_i^{\mathcal{A}}$ and $k_i^{\mathcal{C}} \boxplus_n \texttt{constant}_i^{\mathcal{C}}$. For instance, in Fig. 3, $\tilde{k}_i^{\mathcal{C}}$ is the only hypothesis which could be erroneous due to a carry propagation on its addend. As a result, an attacker should mount one attack taking this carry into consideration, and another one without. This would mean that the total number of collisions would be doubled. Although this selection function may provide some information, we chose to investigate a more efficient attack path.

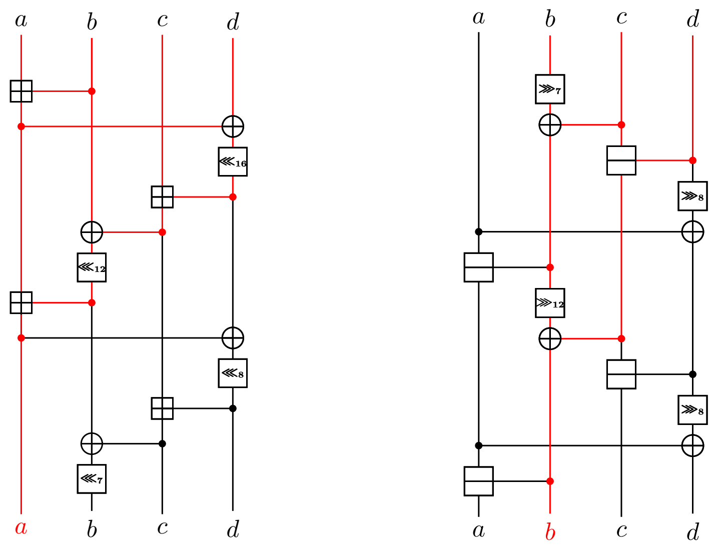
5.3 Benefits of the Reverse Function

The ChaCha QR is trivially invertible and the inverse quarter round (IQR) is defined in Algorithm 4.

Algorithm 4 ChaCha `inv_quarterround(a, b, c, d)`

```
b ⋙= 7;        b ⊕= c;        c ⊟= d;
d ⋙= 8;        d ⊕= a;        a ⊟= b;
b ⋙= 12;       b ⊕= c;        c ⊟= d;
d ⋙= 16;       d ⊕= a;        a ⊟= b;
```

What matters here is that each input word does not have a chance to influence the other three, since the first word does not impact the update of the second one as illustrated in Fig. 5b.



(a) Targeted word at the QR output (b) Targeted word at the IQR output

Fig. 5: Advantage in reversing the QR to define a selection function

Hence, the overall selection can be defined as below

$$\varphi_3\left(b \parallel c \parallel \tilde{d}_i, k_b \parallel k_c\right) = \left(b \boxminus k_b \ggg 7\right) \oplus \left(c \boxminus k_c \ggg 12\right) \oplus \left(c \boxminus k_c \boxminus \tilde{d}_i\right) \quad (6)$$

where $\tilde{d}_i = d_i \boxminus \texttt{nonce}_i$. Regarding the D&C approach where rotations are discarded, it results in the following selection function.

$$\varphi_{4,n}\left(b \parallel c \parallel \tilde{d}_i, k_b^{\mathcal{A}} \parallel k_c^{\mathcal{B}} \parallel k_c^{\mathcal{C}}\right) = \left(b^{\mathcal{A}} \boxminus_n k_b^{\mathcal{A}}\right) \oplus \left(c^{\mathcal{B}} \boxminus_n k_c^{\mathcal{B}}\right) \oplus \left(c^{\mathcal{C}} \boxminus_n k_c^{\mathcal{C}} \boxminus_n \tilde{d}_i^{\mathcal{C}}\right) \quad (7)$$

As less words are involved, the key search space is reduced and still depends on the windows' size.

$$|\mathcal{K}| = \begin{cases} 2^{3n}, & \text{if } n \leq 12 \\ 2^{2n+12}, & \text{if } 12 \leq n \leq 20 \\ 2^{n+32}, & \text{otherwise} \end{cases} \quad (8)$$

However, since the rotations are less pronounced, key words do not overlap if the windows' size does not exceed 12 bits, as depicted in Fig. 6. Throughout the rest of this section, we only consider the case where $n \leq 12$.

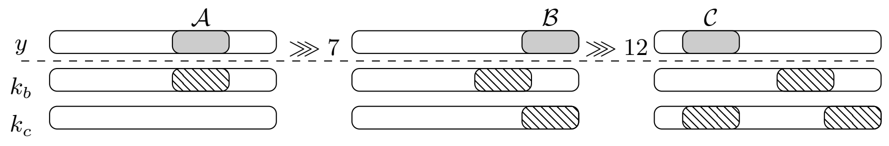


Fig. 6: D&C approach on the ChaCha IQR, $n = 8$

As before, key hypotheses might be affected by carry propagations. However, another advantage of $\varphi_{4,n}$ over $\varphi_{2,n}$ is that one knows the *entire* 32-bit minuend (*i.e.* b or c). Thus, depending on its value, one can calculate the probability of a carry propagation. For instance, when targeting $k_b^{[x,x+n[}$, the probability is

$$p = \mathbb{P}\left(k_b^{[0,x[} > b^{[0,x[}\right) = \frac{2^x - \left(b^{[0,x[} + 1\right)}{2^x} \,. \quad (9)$$

For our simulation with $n = 4$, we took a carry into consideration only if $p > 0.75$. On top of providing a smaller key search space, $\varphi_{4,n}$ is less prone to collisions as shown by our simulation depicted in Fig. 7.

Proposition 2. *An attack on $\varphi_{4,n}$ returns 4 collisions.*

Proof. Flipping the MSB of the minuend/subtrahend also flips the MSB of the modular difference. Therefore, in the case of $\varphi_{4,n}$, flipping the MSB of two n-bit key windows leads to the same output. As a result, the number of collisions is equal to $1 + \binom{3}{2} = 4$. □

This property allows to halve the key search space (*i.e.* $|\mathcal{K}| = 2^{3n-1}$), since all collisions can be retrieved from just one. In the next section, we suggest a more efficient method than repeating this computation over several windows and then sorting the right key from the collisions.

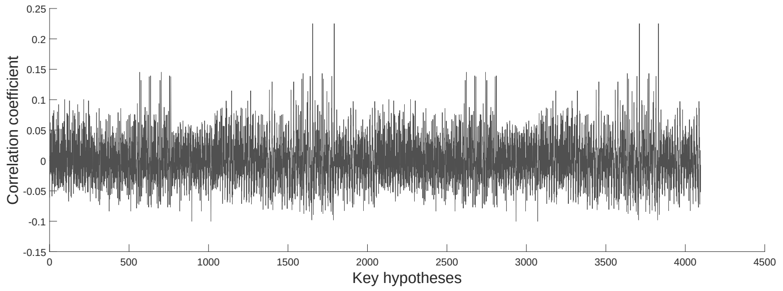


Fig. 7: Attack simulation on $\varphi_{4,4}$

5.4 Overview of the Brickerlayer Attack

Once collisions have been found using $\varphi_{2,n}$ or $\varphi_{4,n}$, one has to reiterate the same procedure on different windows. Instead of executing several attacks in parallel, we suggest to take advantage of windows that have been previously recovered, in order to target larger ones. For instance, once 4 collisions have been found after an attack on $\varphi_{4,n}$, one can target $\varphi_{4,m}$, where $m > n$, with a complexity $|\mathcal{K}| = 2^{3(m-n)+1}$.

Proceeding in this sequential manner has two advantages. First, taking the carry propagation into consideration is only necessary during the first attack. This property is especially interesting for $\varphi_{2,n}$ since there is no way to estimate carry propagations in this case. Second, each attack cancels collisions from the previous ones, since the positions of the collision bits are changed. For instance, regarding $\varphi_{4,n}$ where collisions only depend on MSBs, the bricklayer approach transforms previous collisions into the predictions' lower bits, allowing the correct collision to stand out. This property is less efficient in the case of $\varphi_{2,n}$ since collisions depends on all bits of the n-bit word. Therefore, the correct collision does not stand out directly but some wrong hypotheses are still discarded.

An example application of the bricklayer attack using $\varphi_{4,n}$ is depicted in Fig. 8. Note that from the fourth step, the attack focuses two key windows instead of three because rotations lead to a position that has already been recovered. Finally, the last step considers the entire 32-bit output word using φ_3 and the known bits/collisions.

6 Applications in Practice

6.1 Practical Experiments

All practical experiments presented below were done using an ARM 32-bit Cortex-M3 processor clocked at 24MHz. Note that the DUT does not embed any hardware countermeasure against side-channel attacks. A trigger signal was inserted to indicate the beginning and the end of the penultimate round in order

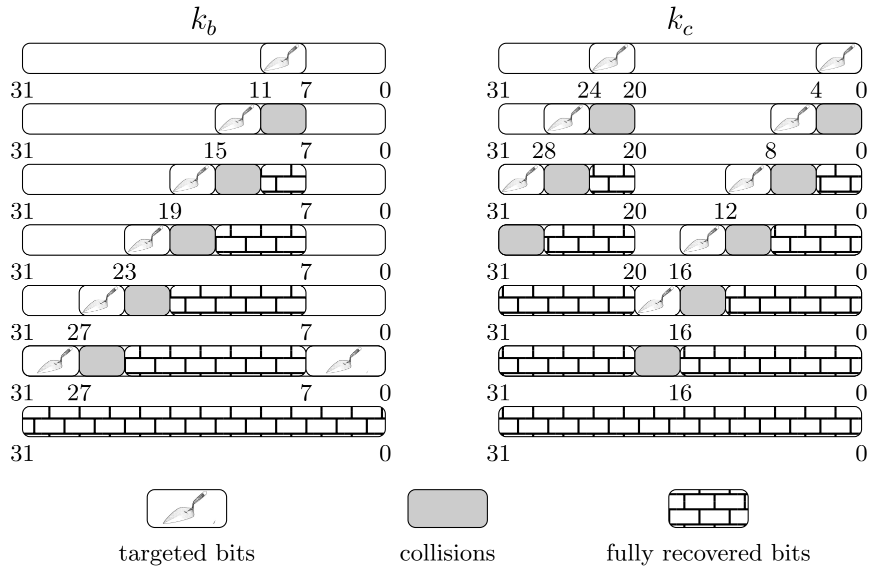


Fig. 8: Bricklayer attack example on IQR

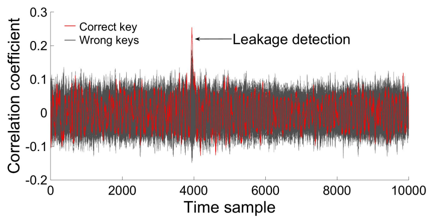
to avoid synchronization complications. EM emanations were measured using a Langer LF-U 5 near-field probe (100kHz-50MHz) and a LeCroy WaveSurfer 10 oscilloscope sampled at 10GS/s. The signal was amplified using a Langer PA 303 BNC preamplifier, providing a gain of 30dB. We used the same leakage model as for our simulations, since our microcontroller leaks the HW of intermediate values.

First, we tried to perform correlation analyses by focusing on arithmetic operations, without success. Figure 9 emphasizes that attacking the final block addition during executions of `quarterround'` was not successfull, whereas for the compiled C version (which stores the intermediate values in RAM), we were able to retrieve the key bits. This reinforced our assumption that, depending on the computing platform, memory accesses can be the only source of exploitable leakage for software implementations.

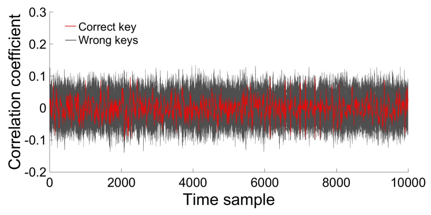
In order to put the bricklayer attack into practice, the following hard-coded input block was used to encrypt 250kB of data, where the counter (*i.e.* $\texttt{nonce}_0$) was incremented for each 512-bit block.

61707865	3320646e	79622d32	6b206574
ad0578e5	e962fc0a	42ffc031	75018bee
b7ae69dc	f1490ca8	89ac12fd	be8466d3
00000000	f1d69cbf	8e34191d	7024af3b

Fig. 10: Input block used for practical experiments



(a) Attack on the final block addition (C compiled)



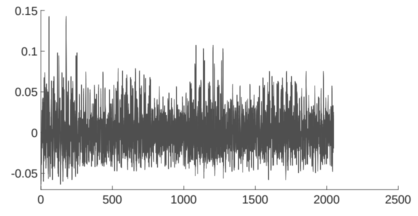
(b) Attack on the final block addition (ARM Assembly)

Fig. 9: Impact of memory accesses on electromagnetic leakage

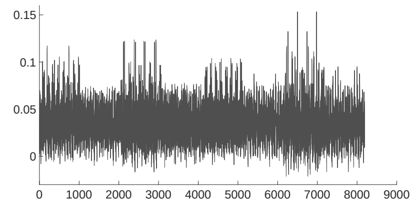
Figure 11 depicts all the correlation curves corresponding to each step of the bricklayer attack when targeting k_2 and k_7. We incremented the windows' length by 4 at each step, exactly as illustrated in Fig. 8, resulting in an overall computational complexity of 2^{13}. All CEMAs were computed by halving the key search space. Consequently, some results do not appear clearly on charts and have to be deduced.

The first step, which targets $k_7^{23\ldots20} \parallel k_7^{3\ldots0} \parallel k_2^{10\ldots7}$, returned the collisions $\mathit{\Gamma} = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4\} = \{56, 176, 2096, 2232\}$. For the next stages, each key hypothesis $\kappa \in \mathcal{K}$ was coupled to each collision $\gamma_j \in \mathit{\Gamma}$ and was placed at the index $i = \kappa \cdot |\mathit{\Gamma}| + j$ of the prediction vector. Thus, higher coefficients at indexes i revealed the correct collision of the previous step γ_j by computing $j = i \bmod |\mathit{\Gamma}|$. Finally, the new collisions are equal to $(i - j) \ / \ |\mathit{\Gamma}|$. For instance, Fig. 11b indicates that the maximum coefficient appears at indexes $i \in \{6499, 6979\}$. Both indexes are congruent to 3 modulo 4, which means that $\gamma_3 = k_7^{23\ldots20} \parallel k_7^{3\ldots0} \parallel k_2^{10\ldots7}$. As a result, the collisions for $k_7^{27\ldots24} \parallel k_7^{7\ldots4} \parallel k_2^{14\ldots11}$ are defined by $\mathit{\Gamma} = \{1624, 1744, 3664, 3800\}$. The remaining steps followed the same methodology, making it possible to recover k_2 and k_7 entirely. Obviously, this can be applied on other IQRs in parallel to recover the whole encryption key.

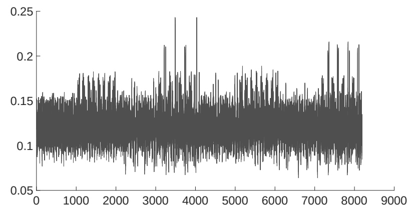
A drawback of the D&C method is the number of required measurements, since the leakage of the omitted bits influences the attacked ones. Thus, more traces are needed in order to average out noise. Figure 12 compares, regarding the number of measurements, an attack on the QR using $\varphi_{2,3}$ with the first step of the bricklayer attack presented above, using the same measurement setup. As a result, to recover the same number of key bits, $\varphi_{4,n}$ requires less traces as it targets larger windows than $\varphi_{2,n}$. However, the number of required traces decreases at each step of the bricklayer attack as the size of targeted windows increases.



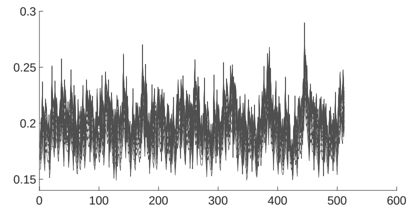
(a) $\kappa = k_7^{23\dots20} \parallel k_7^{3\dots0} \parallel k_2^{10\dots7}$



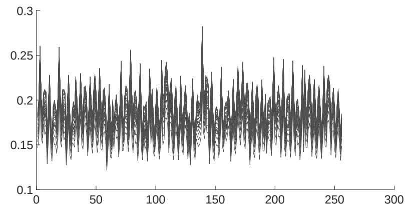
(b) $\kappa = k_7^{27\dots24} \parallel k_7^{7\dots4} \parallel k_2^{14\dots11}$



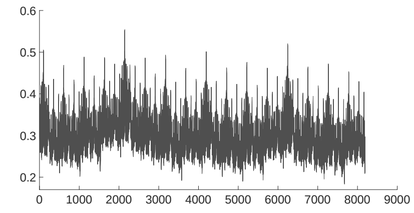
(c) $\kappa = k_7^{31\dots28} \parallel k_7^{11\dots8} \parallel k_2^{18\dots15}$



(d) $\kappa = k_7^{15\dots12} \parallel k_2^{22\dots19}$

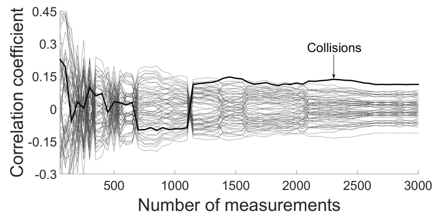


(e) $\kappa = k_7^{19\dots16} \parallel k_2^{26\dots23}$

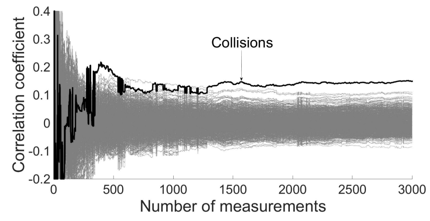


(f) $\kappa = k_2^{31\dots27} \parallel k_2^{6\dots0}$

Fig. 11: CEMAs to recover k_2 and k_7



(a) Attack on the QR using $\varphi_{2,3}$



(b) Attack on the IQR using $\varphi_{4,4}$

Fig. 12: Correlation coefficients to recover 12 key bits

6.2 Attacks' Feasibility on Existing Protocols

In a typical side-channel analysis, it is assumed that the attacker has access to either the plaintext or the ciphertext, but not necessarily to both. In the case of ChaCha, we can consider the knowledge of the nonce as the knowledge of the plaintext. However, attacks using $\varphi_{4,n}$ require the knowledge of the keystream (*i.e.* plaintexts and ciphertexts), in addition to nonces. This is a strong assumption that could be available in an evaluation laboratory but might be hard to set up in practice, leaving the attacks from nonces more realistic. Therefore, we discuss whether the knowledge of nonces is a fair assumption.

By definition, the single requirement for a cryptographic nonce is to be used only once. Therefore, a simple counter could suit the need. However, in cases where many different keys are used, some protocols (*e.g.* TLS) force a part of the nonce (*e.g.* the IV) to be random in order to thwart multi-key attacks [28]. This leaves the block counter as the only predictable part of the nonce. Therefore, if this latter is defined on n bits, then a correlation analysis cannot recover more than $2 \cdot n$ key bits. As a result, it introduces a protocol-level countermeasure which protects a large part of the key.

Still, existing protocols are not defined in this way. For instance, the Secure Shell (SSH) protocol uses the packet sequence number as a 64-bit IV [1] whereas the remaining 64 bits are used for the block counter, which is reset for each packet. Consequently, observing an entire SSH session makes it possible to predict the entire nonce, giving an attacker the opportunity to recover all key words as soon as enough packets are transmitted.

Furthermore, another construction that can be encountered in practice is XChaCha20, which is implemented in the Sodium crypto library [18]. This construction was first proposed for Salsa20 [6] and aims at extending the nonce to 192 bits so that it can be picked at random. The main idea is to encrypt a block with a fixed key k and 128 bits of the random nonce, without executing the final block addition. The first and last 16 bytes of the output result in a 256-bit subkey k'. Finally, the regular ChaCha20 algorithm is executed using the 64 remaining bits of the 192-bit nonce as IV, and k' as encryption key. Note that XChaCha20 is intrinsically resistant against attacks from the keystream, since the final block addition is omitted during the subkey generation. However, the 192-bit nonce must be transmitted in clear and can be entirely known by the attacker.

These real life case studies introduce the need of dedicated countermeasures against side-channel attacks when ChaCha is deployed in such conditions.

7 Towards a Secure Implementation

A common approach to thwart side-channel attacks is the use of *masking*. This countermeasure consists in blinding the processed values x by means of random masks r, so that intermediate variables are impossible to predict. Thus, an attacker has to analyze multiple point distributions, which exponentially increases the attack complexity with the number of shares. In this section, we only discuss first-order masking *i.e.* the case where a single mask is used to randomize the data. Because of their structures, ARX designs need both boolean ($x' = x \oplus r$) and arithmetic ($x' = x \boxminus r$) masking.

To overcome this complication, there are two main approaches. The first one is to switch from one masking scheme to the other whenever necessary. The first conversion algorithms, described by Goubin in [21], have complexity of $\mathcal{O}(1)$ for boolean to arithmetic and $\mathcal{O}(k)$ for arithmetic to boolean, where k refers to the addends' bit size. The latter was then improved by Coron *et al.* to $\mathcal{O}(\log k)$ [15]. The second approach is to directly perform an addition on the masked values, eliminating the need for conversions [23]. However, secure adders usually rely on the recursion formulae involved in arithmetic to boolean conversions. Consequently, they inherit from the same complexity. Some properties of these algorithms are listed in Table 1.

Table 1: Several algorithms' parameters for first-order masking

	Randoms	Auxiliary variables	Operations
Goubin B→A [21]	1	2	7
Goubin A→B [21]	1	3	$5 \cdot k + 1$
Karroumi *et al.* SecAdd [23]	1	3	$5 \cdot k + 8$
Coron *et al.* A→B [15]	2	4	$28 \cdot \log_2 k + 4$
Coron *et al.* SecAdd [15]	3	4	$28 \cdot \log_2 k - 3$

Variants based on precalculations have also been proposed, where time-memory tradeoffs ensure better performances for arithmetic to boolean conversions [17], as well as for secure adders [40].

The best method, in terms of performance, depends on the algorithm to be protected. For instance, masks conversions are more efficient when several arithmetic operations are processed sucessively, since only one arithmetic to boolean conversion is ultimately required. Otherwise, secure adders can lead to better performances as shown by a practical comparison between HMAC-SHA-1 and Speck in [15]. In order to give an insight into the overhead introduced by a first-order masking, we implemented the two secure adders listed in Table 1 in C language, using the same compilation options as described in Sect. 4.3. This allowed us to compare, in terms of performance, our secure implementations of ChaCha20 with the one from the OpenSSL library. Running times given in Table 2 are expressed in clock cycles and were computed with the help of debug sessions. Note that these measurements do not take the generation of random numbers into account since this operation depends a lot on the computing platform. As these countermeasures were implemented in C, they do not ensure the absence of memory accesses within QRs. On the other hand, handling all data in registers during a whole QR may not be possible, since masking also increases memory requirements. Indeed, all algorithms depicted in Table 1 require, in addition to shares, randoms and auxiliary variables. Further investigations need

Table 2: Running time in clock cycles to encrypt a 512-bit block using ChaCha20 on an ARM Cortex-M3

	Time	Penalty factor
ChaCha20 unmasked	4 380	1
ChaCha20 with Karroumi *et al.* SecAdd [23]	121 618	28
ChaCha20 with Coron *et al.* SecAdd [15]	93 993	22

to be carried out to determine which algorithms could minimize memory access within QRs and how to securely manage them.

These practical results point out how difficult it is to effectively secure ARX ciphers' implementations. However, masking is not the only answer to side-channel attacks and is often combined with *hiding* countermeasures. The principle of hiding is to randomize an algorithm execution by running its operations at different moments in time, during each execution [41,37]. This can be achieved by randomly inserting dummy operations and *shuffling*. Shuffling intends to randomly change the sequence of operations that can be computed in arbitrary order. In practice, hiding countermeasures increase the number of traces needed to carry out an attack [34,16].

Regarding ChaCha, operations within a QR cannot be shuffled as they are executed sequentially. On the other hand, each QR can be computed independently from the other, but this is only true for a single round because of switching from column to diagonal rounds. However, there are many ways to implement hiding in practice and further investigations will have to be carried out on the specific case of ChaCha.

8 Conclusions and Further Work

This paper presents side-channel analyses of ChaCha based on leakages related to memory accesses. Our study emphasizes that quantifying the signal available to the attacker at the instruction level could allow to strengthen implementations without much effort.

We compare, from a side-channel point of view, two different software implementations of ChaCha20 on a 32-bit processor. As a result, minimizing memory accesses makes selection functions more complex, to such an extent that they may lead to collisions. We introduce the bricklayer attack to defeat such implementations. Our results show that attacking the reverse QRs (*i.e.* from the keystream) is more efficient than attacking the regular ones (*i.e.* from the input block). However, we highlight that attacks from the input block are the most pragmatic threats since the knowledge of the keystream is a strong assumption.

Finally, we discuss possible countermeasures at several levels and highlight how expensive it is to implement first-order masking for ChaCha20 with practical measurements. Therefore, further work must be undertaken to propose efficient secure implementations of ChaCha.

References

1. chacha20-poly1305@openssh.com Authenticated Encryption Mode (May 2016), http://bxr.su/OpenBSD/usr.bin/ssh/PROTOCOL.chacha20poly1305
2. iOS 10 Security White Paper. Tech. rep., Apple Inc. (March 2017), https://www.apple.com/business/docs/iOS_Security_Guide.pdf
3. Aumasson, J.P., Fischer, S., Khazaei, S., Meier, W., Rechberger, C.: New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba. Cryptology ePrint Archive, Report 2007/472 (2007), http://eprint.iacr.org/2007/472
4. Babbage, S., Borghoff, J., Velichkov, V.: The eSTREAM Portfolio in 2012, http://www.ecrypt.eu.org/ecrypt2/documents/D.SYM.10-v1.pdf
5. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: SIMON and SPECK: Block Ciphers for the Internet of Things. Cryptology ePrint Archive, Report 2015/585 (2015), http://eprint.iacr.org/2015/585
6. Bernstein, D.J.: Extending the Salsa20 nonce, https://cr.yp.to/snuffle/xsalsa-20081128.pdf
7. Bernstein, D.J.: ChaCha, a variant of Salsa20. In: SASC - The State Of The Art Of Stream Ciphers. pp. 273–278 (2008), http://cr.yp.to/chacha/chacha-20080128.pdf
8. Bernstein, D.J.: The Salsa20 Family of Stream Ciphers, pp. 84–97. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-68351-3_8
9. Biryukov, A., Dinu, D., Großschädl, J.: Correlation Power Analysis of Lightweight Block Ciphers: From Theory to Practice, pp. 537–557. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-39555-5_29
10. Boura, C., Lvque, S., Vigilant, D.: Side-Channel Analysis of Grostl and Skein. In: 2012 IEEE Symposium on Security and Privacy Workshops. pp. 16–26 (May 2012), https://www.ieee-security.org/TC/SPW2012/proceedings/4740a016.pdf
11. Brier, E., Clavier, C., Olivier, F.: Correlation Power Analysis with a Leakage Model, pp. 16–29. Springer Berlin Heidelberg, Berlin, Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-28632-5_2
12. Bursztein, E.: Speeding up and strengthening HTTPS connections for Chrome on Android . Tech. rep. (April 2014), https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html
13. Callan, R., Zajić, A., Prvulovic, M.: A Practical Methodology for Measuring the Side-Channel Signal Available to the Attacker for Instruction-Level Events. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 242–254. MICRO-47, IEEE Computer Society, Washington, DC, USA (2014), http://dx.doi.org/10.1109/MICRO.2014.39
14. Choudhuri, A.R., Maitra, S.: Differential Cryptanalysis of Salsa and ChaCha – An Evaluation with a Hybrid Model. Cryptology ePrint Archive, Report 2016/377 (2016), http://eprint.iacr.org/2016/377
15. Coron, J.S., Großschädl, J., Tibouchi, M., Vadnala, P.K.: Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity, pp. 130–149.

Springer Berlin Heidelberg, Berlin, Heidelberg (2015), https://doi.org/10.1007/978-3-662-48116-5_7
16. Couroussé, D., Barry, T., Robisson, B., Jaillon, P., Potin, O., Lanet, J.L.: Runtime Code Polymorphism as a Protection Against Side Channel Attacks. Cryptology ePrint Archive, Report 2017/699 (2017), http://eprint.iacr.org/2017/699
17. Debraize, B.: Efficient and Provably Secure Methods for Switching from Arithmetic to Boolean Masking, pp. 107–121. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), https://doi.org/10.1007/978-3-642-33027-8_7
18. Denis, F.: The XChaCha20-Poly1305 construction, https://download.libsodium.org/doc/secret-key_cryptography/xchacha20-poly1305_construction.html
19. Dworkin, M.J.: SP 800-38A 2001 Edition. Recommendation for Block Cipher Modes of Operation: Methods and Techniques. Tech. rep., Gaithersburg, MD, United States (2001)
20. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic Analysis: Concrete Results, pp. 251–261. Springer Berlin Heidelberg, Berlin, Heidelberg (2001), http://dx.doi.org/10.1007/3-540-44709-1_21
21. Goubin, L.: A Sound Method for Switching between Boolean and Arithmetic Masking, pp. 3–15. Springer Berlin Heidelberg, Berlin, Heidelberg (2001), https://doi.org/10.1007/3-540-44709-1_2
22. Jungk, B., Bhasin, S.: Don't fall into a trap: Physical side-channel analysis of chacha20-poly1305. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2017. pp. 1110–1115 (March 2017)
23. Karroumi, M., Richard, B., Joye, M.: Addition with Blinded Operands, pp. 41–55. Springer International Publishing, Cham (2014), https://doi.org/10.1007/978-3-319-10175-0_4
24. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology. pp. 388–397. CRYPTO '99, Springer-Verlag, London, UK, UK (1999), http://dl.acm.org/citation.cfm?id=646764.703989
25. Kumar, S.V.D., Patranabis, S., Breier, J., Mukhopadhyay, D., Bhasin, S., Chattopadhyay, A., Baksi, A.: A Practical Fault Attack on ARX-like Ciphers with a Case Study on ChaCha20. In: 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC, Taipei, Taiwan (2017)
26. Langley, A., Chang, W., Mavrogiannopoulos, N., Strombergson, J., Josefsson, S.: ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). RFC 7905, RFC Editor (June 2016), http://tools.ietf.org/rfc/rfc7905.txt, http://tools.ietf.org/rfc/rfc7905.txt
27. Lemke, K., Schramm, K., Paar, C.: DPA on n-Bit Sized Boolean and Arithmetic Operations and Its Application to IDEA, RC6, and the HMAC-Construction. In: CHES. pp. 205–219 (2004), http://www.iacr.org/archive/asiacrypt2007/31560191/31560191.pdf
28. Luykx, A., Mennink, B., Paterson, K.G.: Analyzing multi-key security degradation. Cryptology ePrint Archive, Report 2017/435 (2017), http://eprint.iacr.org/2017/435
29. Maitra, S.: Chosen IV Cryptanalysis on Reduced Round ChaCha and Salsa. Discrete Appl. Math. 208(C), 88–97 (Jul 2016), http://dx.doi.org/10.1016/j.dam.2016.02.020
30. Mazumdar, B., Ali, S.S., Sinanoglu, O.: Power analysis attacks on arx: An application to salsa20. In: 2015 IEEE 21st International On-Line Testing Symposium (IOLTS). pp. 40–43 (July 2015)

31. McCann, D., Eder, K., Oswald, E.: Characterising and Comparing the Energy Consumption of Side Channel Attack Countermeasures and Lightweight Cryptography on Embedded Devices. Cryptology ePrint Archive, Report 2015/832 (2015), http://eprint.iacr.org/2015/832
32. McGrew, D., Bailey, D.: AES-CCM Cipher Suites for Transport Layer Security (TLS). RFC 6655, RFC Editor (July 2012), http://tools.ietf.org/rfc/rfc6655.txt
33. Mozaffari-Kermani, M., Azarderakhsh, R.: Reliable Hash Trees for Post-Quantum Stateless Cryptographic Hash-based Signatures. In: 2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS). pp. 103–108 (Oct 2015)
34. Patranabis, S., Roy, D.B., Vadnala, P.K., Mukhopadhyay, D., Ghosh, S.: Shuffling across rounds: A Lightweight Strategy to Counter Side-Channel Attacks. In: 2016 IEEE 34th International Conference on Computer Design (ICCD). pp. 440–443 (Oct 2016)
35. Quisquater, J.J., Samyde, D.: ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards, pp. 200–210. Springer Berlin Heidelberg, Berlin, Heidelberg (2001), http://dx.doi.org/10.1007/3-540-45418-7_17
36. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-tls13-21, Internet Engineering Task Force (Jul 2017), https://tlswg.github.io/tls13-spec/draft-ietf-tls-tls13.html, work in Progress
37. Rivain, M., Prouff, E., Doget, J.: Higher-order masking and shuffling for software implementations of block ciphers. Cryptology ePrint Archive, Report 2009/420 (2009), http://eprint.iacr.org/2009/420
38. Salowey, J., Choudhury, A., McGrew, D.: AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288, RFC Editor (August 2008), http://www.rfc-editor.org/rfc/rfc5288.txt
39. Shi, Z., Zhang, B., Feng, D., Wu, W.: Improved Key Recovery Attacks on Reduced-Round Salsa20 and ChaCha, pp. 337–351. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-37682-5_24
40. Vadnala, P.K., Großschädl, J.: Faster Mask Conversion with Lookup Tables, pp. 207–221. Springer International Publishing, Cham (2015), https://doi.org/10.1007/978-3-319-21476-4_14
41. Veyrat-Charvillon, N., Medwed, M., Kerckhof, S., Standaert, F.X.: Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note, pp. 740–757. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), https://doi.org/10.1007/978-3-642-34961-4_44
42. Yadav, P., Gupta, I., Murthy, S.K.: Study and analysis of eSTREAM cipher Salsa and ChaCha. In: 2016 IEEE International Conference on Engineering and Technology (ICETECH). pp. 90–94 (March 2016)
43. Zohner, M., Kasper, M., Stöttinger, M.: Butterfly-Attack on Skein's Modular Addition, pp. 215–230. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), https://doi.org/10.1007/978-3-642-29912-4_16

A ARM Instructions for Implementations of ChaCha's QR

`-O0` Compilation

```
LDR     r1,[sp,#0x10]
LDR     r0,[sp,#0x00]
ADD     r0,r0,r1
STR     r0,[sp,#0x00]
LDR     r1,[sp,#0x00]
LDR     r0,[sp,#0x30]
EORS    r0,r0,r1
LSLS    r1,r0,#16
LDR     r2,[sp,#0x00]
LDR     r0,[sp,#0x30]
EORS    r0,r0,r2
ORR     r0,r1,r0,LSR #16
STR     r0,[sp,#0x30]
LDR     r1,[sp,#0x30]
LDR     r0,[sp,#0x20]
ADD     r0,r0,r1
STR     r0,[sp,#0x20]
LDR     r1,[sp,#0x20]
LDR     r0,[sp,#0x10]
EORS    r0,r0,r1
LSLS    r1,r0,#12
LDR     r2,[sp,#0x20]
LDR     r0,[sp,#0x10]
EORS    r0,r0,r2
ORR     r0,r1,r0,LSR #20
STR     r0,[sp,#0x10]
LDR     r1,[sp,#0x10]
LDR     r0,[sp,#0x00]
ADD     r0,r0,r1
STR     r0,[sp,#0x00]
LDR     r1,[sp,#0x00]
LDR     r0,[sp,#0x30]
EORS    r0,r0,r1
LSLS    r1,r0,#8
LDR     r2,[sp,#0x00]
LDR     r0,[sp,#0x30]
EORS    r0,r0,r2
ORR     r0,r1,r0,LSR #24
STR     r0,[sp,#0x30]
LDR     r1,[sp,#0x30]
LDR     r0,[sp,#0x20]
ADD     r0,r0,r1
STR     r0,[sp,#0x20]
LDR     r1,[sp,#0x20]
LDR     r0,[sp,#0x10]
EORS    r0,r0,r1
LSLS    r1,r0,#7
LDR     r2,[sp,#0x20]
LDR     r0,[sp,#0x10]
EORS    r0,r0,r2
ORR     r0,r1,r0,LSR #25
STR     r0,[sp,#0x10]
```

`-O3` Compilation

```
LDR     r1,[sp,#0x10]
LDR     r0,[sp,#0x00]
ADD     r0,r0,r1
STR     r0,[sp,#0x00]
LDR     r1,[sp,#0x00]
LDR     r0,[sp,#0x30]
EORS    r0,r0,r1
ROR     r0,r0,#16
STR     r0,[sp,#0x30]
LDR     r1,[sp,#0x30]
LDR     r0,[sp,#0x20]
ADD     r0,r0,r1
STR     r0,[sp,#0x20]
LDR     r1,[sp,#0x20]
LDR     r0,[sp,#0x10]
EORS    r0,r0,r1
ROR     r0,r0,#20
STR     r0,[sp,#0x10]
LDR     r1,[sp,#0x10]
LDR     r0,[sp,#0x00]
ADD     r0,r0,r1
STR     r0,[sp,#0x00]
LDR     r1,[sp,#0x00]
LDR     r0,[sp,#0x30]
EORS    r0,r0,r1
ROR     r0,r0,#24
STR     r0,[sp,#0x30]
LDR     r1,[sp,#0x30]
LDR     r0,[sp,#0x20]
ADD     r0,r0,r1
STR     r0,[sp,#0x20]
LDR     r1,[sp,#0x20]
LDR     r0,[sp,#0x10]
EORS    r0,r0,r1
ROR     r0,r0,#25
STR     r0,[sp,#0x10]
```

Assembly Coded

```
PUSH    {r4-r7}
LDR     r4, [r0]
LDR     r5, [r1]
LDR     r6, [r2]
LDR     r7, [r3]
ADD     r4, r4, r5
EOR     r7, r7, r4
ROR     r7, r7, #16
ADD     r6, r6, r7
EOR     r5, r5, r6
ROR     r5, r5, #20
ADD     r4, r4, r5
EOR     r7, r7, r4
ROR     r7, r7, #24
ADD     r6, r6, r7
EOR     r5, r5, r6
ROR     r5, r5, #25
STR     r4, [r0]
STR     r5, [r1]
STR     r6, [r2]
STR     r7, [r3]
POP     {r4-r7}
```

B Proof of Proposition 1

Proof. We first prove the specific case where $\kappa = \tilde{k}_i^{\mathcal{C}} \parallel k_{i+4}^{\mathcal{B}} \parallel k_i^{\mathcal{B}} \parallel \tilde{k}_i^{\mathcal{A}}$ is such that $\varphi_{2,n}\left(x, \tilde{k}_i^{\mathcal{A}}, k_i^{\mathcal{B}}, k_{i+4}^{\mathcal{B}}, \tilde{k}_i^{\mathcal{C}}\right) = x$ for all x.

Let $\varphi_{2,n}(\cdot, a, b, c, d)$ be the function that maps $x \in \mathbb{Z}_{2^n}$ to $\varphi_{2,n}(x, a, b, c, d)$. From the definition of $\varphi_{2,n}$, we can trivially ensure that if $\varphi_{2,n}(\cdot, a, b, c, d)$ is an identity function on $\mathbb{Z}_{2^n}$, then $\varphi_{2,m}(\cdot, a, b, c, d)$ is an identity function on $\mathbb{Z}_{2^m}$ for $m < n$.

If b, d are both multiples of 2^m for $m \leq n$, the addition terms modulo 2^n do not interfere with the lower m bits. Hence if we split the numbers into lower m and upper $n - m$ bits, *i.e.* $x = \overline{x} \cdot 2^m + \underline{x}$, we can split $\varphi_{2,n}(x, a, b, c, d)$ into two separate actions: $\varphi_{2,n-m}(\overline{x}, \overline{a}, \overline{b}, \overline{c}, \overline{d})$ on $\mathbb{Z}_{2^{n-m}}$ and $\varphi_{2,m}(\underline{x}, \underline{a}, 0, \underline{c}, 0)$ on $\mathbb{Z}_{2^m}$. The latter of these, $\varphi_{2,m}(\underline{x}, \underline{a}, 0, \underline{c}, 0)$ gives the identity function if $\underline{a} = \underline{c}$.

Since we may pick the largest such $m \leq n$ so that 2^m divides b and d, where $m = n$ corresponds to the case when $b = d = 0$ in $\mathbb{Z}_{2^n}$, what remains to be solve is the case where either b or d is odd.

A class of identity functions on $\mathbb{Z}_{2^n}$ is defined by $\varphi_{2,n}(\cdot, 0, b, 0, d)$ where $b \boxplus_n d = 0$. Moreover, XORing with 2^{n-1} has the same effect as adding 2^{n-1}, so we get identify functions $\varphi_{2,n}(\cdot, a, b, c, d)$ for $b, d \in \mathbb{Z}_{2^n}$ and $a, c \in \{0, 2^{n-1}\}$ whenever $a \boxplus_n b \boxplus_n c \boxplus_n d = 0$. Afterwards, XORing x with $2^n - 1$ returns $2^n - 1 - x$ in $\mathbb{Z}_{2^n}$, thus $\varphi_{2,n}(\cdot, 2^n - 1, b, 2^n - 1, d)$ is an identity function when $b = d$. Finally, these two may be combined: $a, c \in \{2^{n-1} - 1, 2^n - 1\}$ makes $\varphi_{2,n}(\cdot, a, b, c, d)$ an identity function if $a \boxplus_n b = c \boxplus_n d$.

In all classes of solution, b and d are either both odd or both even. We focus on the case where they are both odd as the basic solution. We can pick a, c in 8 different ways, either $a, c \in \{0, 2^{n-1}\}$ or $a, c \in \{2^n - 1, 2^{n-1} - 1\}$, an arbitrary odd d, and then solve for b. Thus, there are 2^{n+2} solutions with b, d odd.

For the m lower bits, we have $b' = d' = 0$ and need $a' = c'$ to get an identity function, which gives 2^m solutions. For the upper $n - k$ bits, we get the 2^{n-m+2} solutions found above. We now have to add up the counts for different values of m. For $m = 0, \ldots, n - 2$ this is straightforward. However, the cases $m = n - 1$ and $m = n$ (*i.e.* $b = d = 0$) are both a little different, but are easy to check: they have $3 \cdot 2^n$ and 2^n solutions, respectively. In fact, for these two cases we may use that adding 2^{n-1} is the same as XORing with 2^{n-1}, so $\varphi_{2,n}(\cdot, a, b, c, d)$ becomes an identity function for $b, d \in \{0, 2^{n-1}\}$ if $a \oplus b \oplus c \oplus d = 0$.

All in all, this result in $n \cdot 2^{n+2}$ identity functions.

An apparent advantage of the identity function is that we can make its addition and XOR components independtly zero. If κ does not define an identity function, positions may be more tightly coupled to the others. For example, an attack on $\varphi_{2,3}$ with $\kappa = (000 \parallel 001 \parallel 001 \parallel 000)_2$ only returns 36 collisions. However, the simulation presented in Fig.4 also returns $n \cdot 2^{n+2}$ collisions despite the fact that $\kappa = (00 \parallel 01 \parallel 11 \parallel 11)_2$ does not define an identity function.

Finally, if $n > 4$, there are less than $n \cdot 2^{n+2}$ collisions since some key bits overlap. $\square$