Efficient Side-Channel Protections of ARX Ciphers

Bernhard Jungk[1], Richard Petri[2] and Marc Stöttinger[3]

[1] Fraunhofer Singapore, Singapore, bernhard@projectstarfire.de
[2] Fraunhofer SIT, Germany, richard.petri@sit.fraunhofer.de
[3] Continental Teves, Germany, marc.stoettinger@contiental-corporation.com

Abstract. The current state of the art of Boolean masking for the modular addition operation in software has a very high performance overhead. Firstly, the instruction count is very high compared to a normal addition operation. Secondly, until recently, the entropy consumed by such protections was also quite high. Our paper significantly improves both aspects, by applying the Threshold Implementation (TI) methodology with two shares and by reusing internal values as randomness source in such a way that the uniformity is always preserved. Our approach performs considerably faster compared to the previously known masked addition and subtraction algorithms by Coron et al. and Biryukov et al. improving the state of the art by 36%, if we only consider the number of ARM assembly instructions. Furthermore, similar to the masked adder from Biryukov et al. we reduce the amount of randomness and only require one bit additional entroy per addition, which is a good trade-off for the improved performance. We applied our improved masked adder to ChaCha20, for which we provide two new first-order protected implementations and achieve a 36% improvement over the best published result for ChaCha20 using an ARM Cortex-M4 microprocessor.

Keywords: Modular Addition Masking Side-channel Analysis ChaCha20

1 Introduction

Modular addition is a common operation in many cryptographic algorithms. For instance, ARX ciphers, such as Threefish [FLS+10], Speck [BSS+15], or ChaCha20 [Ber08] rely only on the three operations (modular) addition, rotation and XOR. Software implementations of these ciphers are usually easy to protect against timing side-channel attacks, but at the same time harder to protect against power or EM analysis, e.g. against the butterfly attack on Skein's modular addition [ZKS12] or the bricklayer attack on ChaCha20 [AFM17]. The overhead of applying Boolean masking is high for the addition operation, even with state of the art implementations [BDLCU17, DGLC17]. This is especially evident when comparing ARX ciphers with substitution-permutation network (SPN) ciphers, such as AES, where masked bit-sliced implementations can be used to reduce the overhead significantly [SS16, BGRV15].

In this paper, we investigate how to improve the efficiency of side-channel hardend software implementations of ARX ciphers against power and EM side-channel analysis. To that end, we propose several optimizations to reduce the overhead of such protections. We investigate how to implement the Boolean masking for modular addition and subtraction using a 2-share Threshold Implementation (TI). For our best performing masked addition, most remasking steps of the previous algorithms become redundant and can be removed. It is furthermore possible to optimize the instruction count for ARM implementations using the flexible second operand feature [Ltd18], which allows to execute instructions such

IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925, Vol. 2018, No. 3, pp. 627–653
DOI:10.13154/tches.v2018.i3.627-653

as $z \leftarrow x(y \ll c)$ in one clock cycle, where c is a constant. Our case study for Boolean masking is performed for ChaCha20 on a Cortex-M4 platform. In particular, we provide performance measurements using a the STM32F411 microcontroller [STM14]. We chose ChaCha20 as an example because it is a stream cipher with 256 bit security and hence seems to be a promising candidate for long terms security, because its resilience to quantum computer based attacks with Grover's algorithm.

1.1 Masking Of Modular Addition

Early work on masking of the modular addition suggested to use a conversion technique to switch between different types of masking [Gou01]. In that work, Goubin suggested to use Boolean masking for linear operations and arithmetic masking for modular additions. This setting requires conversions between the two types of masks. While one conversion is practically free with an asymptotic cost of $O(1)$, the conversion from arithmetic to Boolean masks is more complex. Goubin proposed an $O(k)$ algorithm for this task, where k is the bit width of the addition [Gou01]. This asymptotic complexity was later improved to $O(\log k)$ by Coron et al. [CGTV15] using the Kogge-Stone adder (KSA). In addition to the conversion algorithms, Coron et al. also proposed a direct application of Boolean masking to the addition operation, also based on the KSA. In our paper, we focus on the direct application of Boolean masking, but we note, that our work can be also adapted for converting masks.

Several publications followed the work of Coron et al., optimizing the masked modular addition further. Won et al. proposed an improved version of the basic scheme to micro-controllers which have a register width that is smaller than the width of the addition to be implemented [WH17]. Their changes lower the constant of the asymptotic complexity for such targets considerably and can also be applied to our proposed algorithms. Later, Biryukov et al. investigated optimal instruction counts for individual gates, such as SHIFT, AND and XOR [BDLCU17]. Using these optimized masked gates to implement the addition operation, Biryukov et al. achieved a considerable performance improvement over Coron et al.'s masked addition. Dinu et al. also reduced the constant in the asymptotic complexity using a carry save adder [DGLC17]. However, their algorithm has a variable runtime and hence, it might be susceptible to timing-based side-channel attacks. Another development by Schneider et al. transferred the masked addition to hardware implementations using a Threshold Implementation approach with three shares [SMG15]. While the number of clock cycles is reduced significantly to only six for a first-order secure hardware implementation of a 32 bit addition, a serialized software implementation would need considerably more clock cycles due to the processing with three shares instead of only two.

Despite the recent progress, the overall overhead to apply masking to modular addition is still high, which results in a considerable total runtime overhead. For instance, in [AFM17] an overhead of more than 21× is reported for ChaCha20 on ARM Cortex-M4 platforms using the masked addition from Coron et al. [CGTV15]. Slightly different penalty factors between 11× and 22× have been reported for other algorithms such as Speck-64/128 in [BDLCU17].

Another drawback of most published first-order masking schemes for modular addition is, that they need a high amount of additional randomness. While Coron et al. state in [CGTV15], that some masks can be reused for other additions, reuse can lead to other issues as mentioned in [MOP07]. For instance, collision correlation attacks could be possible [RL13]. Consequently, mask reuse has to be limited for security sensitive applications and a conservative approach introduces new masks regularly during the computation, e.g. one new mask per addition. This adds a considerable overhead, especially if the entropy source of a system does not supply a sufficient amount of random bits fast enough.

The exception to these additional randomness requirements is the work of Biryukov et al., which reduces the cost of randomness to zero by optimizing the masked AND gate

[BDLCU17]. However, according to the authors, their implementation of the masked AND gate is not easily composable in the classical Boolean masking setting. Hence, a mask refresh with the original input masks has to be used to guarantee the uniformity, which is a basic requirement for first-order masking schemes. This still adds some runtime overhead to the implementation.

1.2 Our Contributions

In this paper, we study improvements on Boolean masking for ARX ciphers. For Boolean masking, we use Threshold Implementations (TI) as our basis [NRR06, NRS08]. Originally, TI has been designed as an alternative to traditional Boolean masking schemes for hardware implementations. The main feature of TI is its security in the presence of glitches [NRR06, NRS08]. In its original form, it uses $d + 1$ shares, where d is the degree of the function to share. However, many papers have proposed to decompose functions with a higher degree into functions with a smaller degree [BNN+12, KNPW13]. In the best case, this results in degree $d = 2$ and hence, three shares seemed to be the minimum number of shares needed for a first-order secure TI.

However, recent generalizations of the first-order secure TI scheme have reduced the minimum of three shares to only two [RBN+15, CFE16, DCRB+16]. As a side-effect, the runtime overhead in software implementations can be significantly reduced. Therefore, it is worthwhile to revisit TI in the software setting. In this work, we demonstrate that the number of elementary operations (AND, SHIFT, XOR, etc.) for the masked addition operation can be reduced compared to the current state of the art by Biryukov et al. [BDLCU17].

We discovered two main optimization possibilities. Firstly, we are able to demonstrate that some of the remasking steps introduced in the masked addition proposed in [BDLCU17] are unnecessary in our improved algorithm. Secondly, if we include the flexible second operand feature of the ARM instruction set in our study, the number of operations can be reduced without decreasing the theoretical security. In total, we reduce the number of ARM instructions from $22 \cdot \log_2 k + 6$ [BDLCU17] to $14 \cdot \log_2 k + 6$. We also study the modular subtraction used for some decryption algorithms (e.g. for the decryption in Speck) and show that a much simpler algorithm, based directly on the two's complement representation, can outperform the proposed algorithm from [BDLCU17] by more than 50%, reducing the number of ARM instructions from $30 \cdot \log_2 k + 6$ [BDLCU17] to only $14 \cdot \log_2 k + 13$.

To reduce the amount of additional randomness, we investigate the following two ideas. Firstly, we show how to apply a simple but effective scheme to reuse randomness, similar to the Changing of the Guards technique proposed by Daemen [Dae17]. In contrast to Daemen's work, our technique can also be applied to modular addition. The difficulty here is, that modular addition itself is not a permutation and therefore, the earlier proof technique by Daemen does not apply. With our proof, the application to modular addition becomes straightforward and we can prove that the uniformity is preserved for the modular addition and also for full implementations of ARX ciphers. Secondly, we investigate the use of the randomness-free masking of the AND gate described by Biryukov et al. [BDLCU17]. We show, that most remasking steps in the masked addition as proposed in [BDLCU17] are unnecessary and thus, they can be removed, if their improved formula for the AND gate is integrated in our design. The exception is the initial computation of the shared propagates (p_0, p_1) and the generates (g_0, g_1). When applying the masked AND gate to compute the generates, the distributions of the shared propagates and generates are not independent and hence, the operations in the for loop would show some leakage. We solve this issue with a one-time refreshing of the shares, which makes the distributions of the generates and propagates in the inner loop independent of each other, consequently removing the leakage in an efficient way.

Using those two techniques, we reduce the required additional randomness to *one* bit for each execution of ChaCha20 (and most other ARX ciphers). This reduces the amount of consumed uniform randomness significantly and thus, further increases the efficiency of implementations protected by Boolean masking.

Overall, our efforts to improve the theoretical performance of the masked addition operation show significant improvements. However, we caution that in practice a software implementation of the proposed algorithms is challenging, since the independent leakage assumption does not hold for most microprocessors. For example, the Cortex-M3 and Cortex-M4 microprocessors are well known to leak information due to their pipeline registers [CGD17] and also due to other effects. These problems are not unique to our algorithms, but are also present in most other previously presented implementations of Boolean masking in software, as implementing masking securely is very challenging [BGG+14, BGRV15, CGD17, PV17].

1.3 Paper Organization

The remainder of the paper is organized as follows. In Section 2, we introduce the Kogge-Stone adder as our main addition algorithm. Then, we discuss side-channel countermeasures for ARX ciphers in Section 3. In this section, we first recap the earlier proposed Boolean masking scheme for modular addition (Section 3.2) and then move on to introduce our new 2-share TI scheme (Section 3.3). The performance numbers for the Cortex-M4 implementations are given in Section 4. We conclude the paper in Section 5 with an outlook for further research.

2 Modular Addition and Subtraction Algorithms

The asymptotically most efficient adders for adding two variables are (parallel) prefix adders [LF80]. For our work, the Kogge-Stone adder (KSA) [KS73] is in particular interesting, as shown in previous work by Coron et al. [CGTV15]. Like all prefix adders, the KSA has a logarithmic runtime complexity in terms of the bit width k, when implemented in software. While the asymptotic runtime is $O(\log k)$ for all prefix adders, the absolute number of operations in a software implementation may vary greatly, because some of the prefix adders would require a bit-wise processing in software, which would be very inefficient. However, in the case of the KSA, the structure lends itself well to software implementations. As shown in Algorithm 1, each loop iteration of the KSA consists of several shifts, ANDs and XORs. This is attractive for a Boolean masking approach,

Algorithm 1 Kogge-Stone Adder (KSA)

Require: $x, y \in \mathbb{Z}_{2^k}$, $k > 0$
Ensure: $z = (x + y) \bmod 2^k$
1: $n \leftarrow \max(\lceil \log_2(k-1) \rceil, 1)$
2: $g \leftarrow x \wedge y$
3: $p \leftarrow x \oplus y$
4: **for** $i = 1$ to $n - 1$ **do**
5: $\quad g \leftarrow (p \wedge (g \ll 2^{i-1})) \oplus g$
6: $\quad p \leftarrow (p \wedge (p \ll 2^{i-1}))$
7: **end for**
8: $g \leftarrow (p \wedge (g \ll 2^{n-1})) \oplus g$
9: $z \leftarrow x \oplus y \oplus 2g$
10: **return** z

Algorithm 2 KSA-based Subtraction

Require: $x, y \in \mathbb{Z}_{2^k}$, $k > 0$
Ensure: $z = (x - y) \bmod 2^k$
1: $n \leftarrow \max(\lceil \log_2(k-1) \rceil, 1)$
2: $\boldsymbol{y \leftarrow \bar{y}}$
3: $g \leftarrow x \wedge y$
4: $p \leftarrow x \oplus y$
5: $\boldsymbol{g \leftarrow g \oplus (p \wedge 1)}$
6: **for** $i = 1$ to $n - 1$ **do**
7: $\quad g \leftarrow (p \wedge (g \ll 2^{i-1})) \oplus g$
8: $\quad p \leftarrow (p \wedge (p \ll 2^{i-1}))$
9: **end for**
10: $g \leftarrow (p \wedge (g \ll 2^{n-1})) \oplus g$
11: $z \leftarrow x \oplus y \oplus 2g \boldsymbol{\oplus 1}$
12: **return** z

because all k bits of the addition are processed in parallel,[1] which reduces the cost of an implementation significantly compared to a bit-wise implemented masking scheme.

Some cryptographic algorithms do not only require the modular addition operation, but also subtraction. The easiest way to implement subtraction is to reuse the addition operation and rely on a two's complement representation, as depicted in Algorithm 2. A masked implementation of the described subtraction algorithm is much faster than the one proposed in [BDLCU17], since there are only a couple of additional operations to mask compared to the addition.

The improved subtraction algorithm works based on the observation that subtraction is equivalent to addition with the two's complement of the second operand. Hence, a straightforward subtraction using the two's complement involves a bit-wise inversion and an addition of 1. Thus, because of the addition, a direct application of this idea would be inefficient. The same behavior can be achieved, by only computing the bit-wise complement before the addition and instead of adding 1 to the second operand, the carry-in is set to $c_0 = 1$. These changes result in only minor difference to the addition algorithm. To show that this indeed produces the correct results, we look at how prefix adders compute the carry bits c_i from generates g_i and propagates p_i [Ros60]:

$$
\begin{aligned}
c_0 &\leftarrow c_0 \\
c_1 &\leftarrow g_0 \oplus p_0 c_0 \\
c_2 &\leftarrow g_1 \oplus p_1 g_0 \oplus p_1 p_0 c_0 \\
c_3 &\leftarrow g_2 \oplus p_2 g_1 \oplus p_2 p_1 g_0 \oplus p_2 p_1 p_0 c_0 \\
c_4 &\leftarrow g_3 \oplus p_3 g_2 \oplus p_3 p_2 g_1 \oplus p_3 p_2 p_1 g_0 \oplus p_3 p_2 p_1 p_0 c_0
\end{aligned}
$$

We can see, that in each equation only the last term includes c_0.

With the exception of c_0 and c_1, the inner loop of the KSA computes all carries, i.e. at the end of the loop, the generates are exactly the carry bits. Therefore, we can slightly change the initialisation of g before the for loop. In particular, we set $c_0 = 1$ and hence, have to compute $g_0 \leftarrow g_0 \oplus p_0$ before the loop, which already produces c_1 before the first loop iteration (Line 5). Then, the loop (Lines $7 - 8$) produces

$$g_1 \leftarrow g_1 \oplus p_1 g_0 \oplus p_1 p_0$$

[1] Up to the data register width of the processor.

in the first iteration,

$$g_2 \leftarrow g_2 \oplus p_2 g_1 \oplus p_2 p_1 g_0 \oplus p_2 p_1 p_0$$
$$g_3 \leftarrow g_3 \oplus p_3 g_2 \oplus p_3 p_2 g_1 \oplus p_3 p_2 p_1 g_0 \oplus p_3 p_2 p_1 p_0 c_0$$

in the second iteration, g_4, g_5, g_6, g_7 in the third iteration and so on. After loop iteration $\log_2(k)$, $c_1, \ldots, c_k$ will be equal to $g_0, \ldots, g_{k-1}$.

The KSA algorithm works on whole words and not individual bits. Therefore, it is necessary to additionally compute the AND with 1 to obtain the initial $g = g \oplus p_0$ (Algorithm 2, Line 5). We also have to take care of the carry-in $c_0 = 1$ when computing the final output, because g will only contain c_1 to c_{k-1} and not c_0. Therefore, we have to change the final computation of the output z to include the operation $\oplus 1$ (Line 11). Overall, we add only four operations for the subtraction compared to the basic addition operation (marked with boldface in Algorithm 2, Lines 2, 5, and 11).

3 Side-Channel Countermeasures

In our paper, we study the efficiency of first-order DPA countermeasures for modular addition and also for the integration of the modular addition operation in the implementation of an ARX cipher. We first discuss existing masking countermeasures of Boolean masking for the modular addition proposed by Coron et al. [CGTV15] and optimized versions [BDLCU17, DGLC17]. Then, we develop a computationally more efficient masking countermeasure, based on the Threshold Implementation (TI) scheme with two shares [NRR06, NRS08, RBN+15, CFE16].

The proposed optimizations significantly reduce the instruction count over the previously best constant-time masking scheme for modular addition in [BDLCU17]. To the best of our knowledge, this is the first report of showing that 2-share TI may outperform classical Boolean masking schemes on software platforms. There are two sources for this improvement. Firstly, our best algorithm does not require most of the remasking steps for the internal values during the addition operation as in the scheme by Biryukov et al. [BDLCU17]. Secondly, we show that the flexible second operand feature of the ARM instruction set can reduce the instruction count further. Overall, we achieve a significant improvement over the proposed algorithms in [BDLCU17] (Table 2).

Our first masked addition algorithm only needs one bit of uniform randomness as input, compared to the algorithm in [BDLCU17]. We will call this additional input *guard share* in the remainder of this paper, like in Daemen's Changing of the Guards paper [Dae17]. We show formally how to use this guard share together with the reuse of randomness from other shares to achieve a uniform first-order secure TI of the KSA. Additionally, our addition algorithm also generates an output guard share, which can be used as an input share for the next addition in an ARX cipher. Therefore, we can chain several additions without the need to sample new entropy during the computation of the entire ARX cipher. We can prove that one bit of entropy is needed per en- or decryption operation.

In addition, our second proposed masked addition exploits the optimized masked AND gate in [BDLCU17]. Using this optimized AND gate, we can remove all but one mask refreshings without destroying the uniformity property. The single mask refreshing is necessary after computing the initial propagates and generates, because the distributions are not independent of each other. However, after a one-time refreshing after the first AND gate is sufficient to remove the problematic dependencies from the modular addition algorithm with a low overhead per addition.

3.1 Overview of Masked Addition

ARX ciphers mix linear operations with non-linear modular additions. As noted in [CG00], it is easy to apply Boolean masking to linear operations, but the addition operation is simpler to protect using arithmetic masking. Unfortunately, the masks used for the two masking schemes are incompatible with each other and hence, the masks have to be converted. In [Gou01], Goubin proposed the first approach to convert Boolean to arithmetic masks and vice versa. A subsequent publication by Coron et al. reduced the conversion cost from Boolean to arithmetic masks from $O(k)$ to $O(\log k)$ [CGTV15]. In the same paper, Coron et al. also investigated a direct application of Boolean masking to the Kogge-Stone adder as an alternative to the mask conversion. The performance of both methods is similar, but depending on the number of subsequent additions without mask conversion, either the mask conversion or the direct application of Boolean masking is faster. Since our main study object – ChaCha20 – would need a conversion after each addition, we concentrate on the direct application of Boolean masking, because mask conversion would incur a higher overhead. However, our techniques are also applicable to the case of mask conversion, with minor differences.

3.2 Conventional Boolean Masking of the KSA

The application of Boolean masking to the KSA algorithm can be performed by implementing secured versions of the AND, XOR, and SHIFT operations. However, instead of processing the addition bit-wise, parallel versions of these operations can be used to implement the KSA. The secured versions of the gates are `SecAnd`, `SecXor` and `SecShift` as proposed in [CGTV15, BGRV15]. We also checked the application to our subtraction variant, which improves on the one proposed in [BGRV15], reducing the cost of subtraction to $22 \log_2 k + 10$. Both the masked addition and the masked subtraction can be found in Appendix A.

The main drawback in both algorithms are the remaskings in the inner loop after `SecShift` and `SecAnd` (Algorithm 12, Lines 4, 10, and 12, Algorithm 13, Lines 5, 12, 14). Also, it is not easy to integrate the shift operation into the other Boolean operations using the flexible second operand feature. Both issues can be resolved by constructing the Boolean masking in a different way, using 2-share TI.

A competing approach to the KSA-based masked adders is based on the carry-save addition (CSA) [DGLC17]. The main benefit of masking the carry-save adder is, that on average, the loop of the carry-save addition terminates quite early and therefore, it is possible to reduce the average number of clock cycles. However, this approach introduces a timing side-channel, which may be possible to exploit. Furthermore, compared to the state of the art masked addition by Biryukov et al., the performance gain is only about 7%. Also our fastest proposed algorithm beats the published CSA-based results without compromising on the security by exposing a potential timing side-channel. Hence, we do not further discuss this masked addition in this paper.

3.3 2-Share Threshold Implementation of the KSA

Threshold Implementations (TI) have been introduced in [NRR06, NRS08] as an alternative to Boolean masking schemes. The main improvement of TI over most conventional masking schemes is a security proof, which shows that TI-based implementations are also secure in the presence of glitches.

TI is based on a linear application of Shamir's secret sharing scheme, i.e. the original data is divided into multiple shares and the computation is performed on these shares. For software implementations, TI has not been sparked much interest, because of the original requirement to have a minimum of three shares for a first order secure implementation

[NRR06, NRS08], compared to only two shares for normal Boolean masking. However, recently, it has been shown how to construct a secure TI-like sharing with only two input shares and four output shares. These four output shares are then collapsed to two input shares in the following clock cycle [RBN+15, CFE16]. This reduces the area overhead for hardware implementations [CFE16, DCRB+16] and as we show in our paper, it also has a potential application to improve the performance of masked software implementations. Implementations which use the TI scheme have to fulfill the following three basic properties:

1. *Correctness* requires that the linear combination of all input and output shares is equal to the correct original value of the unshared input or output. However, it is not necessary, that all intermediate states that are generated during the computation of the output shares have this property.

2. *Non-completeness* requires that a sharing with d input shares recombines at most $d-1$ shares to be first-order secure. For fulfilling this requirement, a two share implementation of a non-linear gate needs to produce more than two output shares first and save the intermediate results to registers. After registering the intermediate outputs, it is possible to collapse the shares to the original number of two shares. This also means that at least two clock cycles are necessary to achieve non-completeness with only two shares.

3. *Uniformity* requires that all inputs and outputs of a shared function are uniformly shared. Therefore, it is sufficient to show that, if the input shares are a uniform sharing, then the output shares are also a uniform sharing. However, it is not necessary that all intermediate states of the shared function are uniform sharings, because if the inputs are always uniform sharings, then the distributions of internal states will be independent of the original unshared input.

For software implementations as in this paper, achieving non-completeness is easier than for parallel hardware implementations, because each operation is stored to a register anyway. Therefore, if none of the individual terms recombines d shares of the same variable before the register write and if all of the input shares are independent uniform sharings, non-completeness is always fulfilled. However, this is not enough to achieve a secure software implementation, because the independent leakage assumption often does not hold due to other physical defaults [BDF+17], e.g. register reuse, leading to transition-based leakage [BDF+17] or coupling, which might recombine shares non-linearly [DCBG+17]. For simplicity, we assume in the following proofs, that no glitches, no distance-based leakages and no coupling effects occur, i.e. that the independent leakage assumption holds.

We would like to point out, that we follow a less strict interpretation of non-completeness for our sharings as previously proposed in [RBN+15, CFE16], i.e. in a two share implementation with $x = x_0 \oplus x_1$ and $y = y_0 \oplus y_1$, the pair (x_0, y_1) (respectively (x_1, y_0)) can be combined in a single term, as long as x_0 and y_1 (respectively x_1 and y_0) are independently distributed of each other. Otherwise, with only two shares, there would be no possible sharing for any non-linear function. As proposed in [RBN+15, CFE16], we have to increase the number of output shares to four shares. After an optional mask refresh, the four output shares can be collapsed to two shares, such that the number of shares does not increase by each operation.

In this setting, we first consider a single AND gate computing $z = x \wedge y$. The sharing can be performed as follows, using a direct sharing approach [BNN+12, CFE16, GMK16]:

$$s_0 \leftarrow x_0 \wedge y_0, \qquad s_1 \leftarrow x_0 \wedge y_1, \qquad s_2 \leftarrow x_1 \wedge y_0, \qquad s_3 \leftarrow x_1 \wedge y_1$$

Since this leaves us with four output shares, we have to recombine some of the shares, e.g. $z_0 \leftarrow s_0 \oplus s_3$ and $z_1 \leftarrow s_1 \oplus s_2$.

Unfortunately, the sharing (z_0, z_1) is not uniform. Therefore, we have to repair the uniformity by refreshing some of the shares with fresh uniform randomness (Equation 3). This can be achieved using one additional guard share m and then, by computing the following sequence of operations [GMK16]:

$$s_0 \leftarrow x_0 \wedge y_0, \qquad s_1 \leftarrow x_0 \wedge y_1 \tag{1}$$
$$s_2 \leftarrow x_1 \wedge y_0, \qquad s_3 \leftarrow x_1 \wedge y_1 \tag{2}$$
$$t_0 \leftarrow s_0 \oplus m, \qquad t_1 \leftarrow s_1 \oplus m \tag{3}$$
$$z_0 \leftarrow t_0 \oplus s_2, \qquad z_1 \leftarrow t_1 \oplus s_3 \tag{4}$$

Lemma 1. *The output sharing (z_0, z_1) computed by Equations 1 to 4 is a correct, non-complete and uniform sharing of $z = xy$, if (x_0, x_1), (y_0, y_1) are uniform sharings of x and y, if the distribution of m is independent of (x_0, x_1) and (y_0, y_1), and if the intermediate state (t_0, t_1) is stored to a register before computing the output sharing (z_0, z_1).*

Proof. The correctness of the sharing defined by Equations 1 and 2 follows from the construction of the first part by direct sharing. Furthermore, we add the guard share m to two of the four shares, which means that this operation will not change the correctness of the sharing, since m is canceled out when retrieving the unshared value of z. The last step, collapsing four shares to two shares also does not change the correctness, because of the application of only a linear operation.

The non-completeness property holds, because firstly, every operation has two uniformly shared inputs and secondly, we assume that (t_0, t_1) is registered before the output shares (z_0, z_1).

The proof of uniformity can be obtained by enumerating all possibilities. We used the Python script in the appendix for verification (Listing 1 in Appendix C). □

In a similar fashion also $z = xy \oplus u$ can be shared as follows, where u_0 and u_1 takes the place of m and hence, no refreshing is needed [CFE16]. In the remainder of this paper, this gate is called AND-XOR gate.

$$s_0 \leftarrow x_0 \wedge y_0, \qquad s_1 \leftarrow x_0 \wedge y_1 \tag{5}$$
$$s_2 \leftarrow x_1 \wedge y_0, \qquad s_3 \leftarrow x_1 \wedge y_1 \tag{6}$$
$$t_0 \leftarrow s_0 \oplus u_0, \qquad t_1 \leftarrow s_1 \oplus u_1 \tag{7}$$
$$z_0 \leftarrow t_0 \oplus s_2, \qquad z_1 \leftarrow t_1 \oplus s_3 \tag{8}$$

The proof of the TI properties for the sharing of $z = xy \oplus u$ is essentially the same as for Lemma 1, hence, we skip the proof of the following lemma. However, the proof of uniformity is slightly different, therefore, we supply a slightly adapted Python script in Listing 2 in Appendix D.

Lemma 2. *The output sharing (z_0, z_1) computed by Equations 5 to 8 is a correct, non-complete and uniform sharing of $z = xy \oplus u$, if (x_0, x_1), (y_0, y_1) are uniform sharings of x and y, if the distribution of the sharing (u_0, u_1) is independent of (x_0, x_1) and (y_0, y_1), and if the intermediate state (t_0, t_1) is stored to a register before computing the output sharing (z_0, z_1).*

These two sharings are essential for the 2-share TI of our masked modular addition operation. However, a direct application to the KSA algorithm leads to a very high consumption of randomness, because of the necessary mask refreshs in each loop iteration. We can solve this in an elegant way by reusing shares.

We apply the reuse scheme to a parallel execution of k shared AND gates, where each individual AND gate has a different set of inputs. We show that it can be shared uniformly with only a single guard share, instead of k uniformly distributed random refresh masks.

Algorithm 3 2-Share TI of a k-radix AND gate (`SecAnd`).

Require: $x_0, x_1, y_0, y_1 \in \mathbb{Z}_{2^k}$, $k > 0$, $u \in \{0, 1\}$
Ensure: $(z_0 \oplus z_1) = (x_0 \oplus x_1) \wedge (y_0 \oplus y_1) \bmod 2^k$
1: $m \leftarrow ((x_0 \gg 1) \oplus (u \ll (k-1)))$ # Generate refresh mask
2: $(s_0, s_1, s_2, s_3) \leftarrow (x_0 \wedge y_0, x_0 \wedge y_1, x_1 \wedge y_0, x_1 \wedge y_1)$ # Shared AND
3: $(t_0, t_1) \leftarrow (s_0 \oplus m, s_1 \oplus m)$ # Refresh masks
4: $(z_0, z_1) \leftarrow (t_0 \oplus s_2, t_1 \oplus s_3)$ # Collapse shares
5: **return** (z_0, z_1)

Algorithm 4 2-Share TI of a k-radix AND-XOR gate (`SecAndXor`).

Require: $x_0, x_1, y_0, y_1, u_0, u_1 \in \mathbb{Z}_{2^k}$, $k > 0$
Ensure: $(z_0 \oplus z_1) = ((x_0 \oplus x_1) \wedge (y_0 \oplus y_1)) \oplus (u_0 \oplus u_1) \bmod 2^k$
1: $(s_0, s_1, s_2, s_3) \leftarrow (x_0 \wedge y_0, x_0 \wedge y_1, x_1 \wedge y_0, x_1 \wedge y_1)$ # Shared AND
2: $(t_0, t_1) \leftarrow (s_0 \oplus u_0, s_1 \oplus u_1)$ # Shared XOR
3: $(z_0, z_1) \leftarrow (t_0 \oplus s_2, t_1 \oplus s_3)$ # Collapse shares
4: **return** (z_0, z_1)

In this scheme, we reuse the property that inputs are uniformly distributed and thus, the inputs to an adjacent AND gate can be used to guard the uniformity of its (direct) neighbor. Only the uniformity of one shared AND gate needs to be protected by an additional independently uniformly distributed random input.

Based on this idea, we develop Algorithm 3 (`SecAnd`) for a k-radix AND Gate and a corresponding proof of the TI properties (Lemma 3). Additionally, in Algorithm 4 (`SecAndXor`) we show the k-radix variant of the AND-XOR gate. We skip the proof of the TI properties for Algorithm 4, because it is only a repeated application of Lemma 2. In all of the following proofs, we assume a certain bit order, i.e. the lowest significant bit is bit 0 and the highest significant bit is bit $k-1$.

Lemma 3. *Algorithm 3 implements a correct, non-complete and uniform sharing of a k-radix AND gate with k inputs and k outputs,*

1. *if the inputs $(x_0, x_1), (y_0, y_1)$ are uniform sharings of x, y,*
2. *if u is uniformly distributed,*
3. *if the distributions of $x_0[k-1], x_1[k-1], y_0[k-1], y_1[k-1]$ are independent of the distribution of u,*
4. *and if the intermediate state (t_0, t_1) is stored to a register before computing the output sharing (z_0, z_1).*

Proof. By Lemma 1, the computation of the top most bits $k-1$ of z_0 and z_1, i.e. the pair $(z_0[k-1], z_1[k-1])$, with $m[k-1] = u$ is a correct, non-complete and uniform sharing, because the mask $m[k-1] = u$ is assumed to be independent of $x_0[k-1]$, $x_1[k-1]$ and $y_0[k-1]$, $y_1[k-1]$. Correctness and non-completness for the bits 0 to $k-2$ can be also directly derived from Lemma 1.

For the uniformity of the bits 0 to $k-2$, we have to show that the guard share is uniformly distributed and independent from the input shares. The uniformity of the guard shares follows from the reuse of the uniformly distributed inputs $x_0[1], \ldots, x_0[k-1]$ which are reused as guard shares. Furthermore, the distributions of the guard shares are independent of the inputs, because different bits of x_0 are used, i.e. $m[i] = x_0[i+1]$ for $0 \le i \le k-2$, whereas the inputs to the shared AND gates are $x_0[i]$, $x_1[i]$ and $y_0[i]$, $y_1[i]$ for $0 \le i \le k-2$, which are, by assuming uniform sharings, all independently distributed from $x_0[i+1]$. Hence, uniformity for the shared k-radix AND follows. □

Algorithm 5 2-Share TI of a k-radix AND-SHIFT gate (`SecAndShift`).

Require: $x_0, x_1 \in \mathbb{Z}_{2^k}$,k>0, $u \in \{0,1\}$, $0 \leq i < k$
Ensure: $(z_0 \oplus z_1) = (x_0 \oplus x_1) \wedge ((x_0 \oplus x_1) \ll i) \bmod 2^k$

$m \leftarrow ((x_0 \gg 1) + (u \ll (k-1)))$	# Generate refresh mask
$(y_0, y_1) \leftarrow (x_0 \ll i, x_1 \ll i)$	# Shared SHIFT
$(s_0, s_1, s_2, s_3) \leftarrow (x_0 \wedge y_0, x_0 \wedge y_1, x_1 \wedge y_0, x_1 \wedge y_1)$	# Shared AND
$(t_0, t_1) \leftarrow (s_0 \oplus m, s_1 \oplus m)$	# Refresh masks
$(z_0, z_1) \leftarrow (t_0 \oplus s_2, t_1 \oplus s_3)$	# Collapse shares
return (z_0, z_1)	

Algorithm 6 2-Share TI of a k-radix AND-XOR-SHIFT gate (`SecAndShiftXor`).

Require: $x_0, x_1, y_0, y_1 \in \mathbb{Z}_{2^k}$, $k > 0$, $u \in \{0,1\}$, $0 \leq i < k$
Ensure: $(z_0 \oplus z_1) = (x_0 \oplus x_1) \wedge ((y_0 \oplus y_1) \ll i) \oplus y_0 \oplus y_1) \bmod 2^k$

1: $(v_0, v_1) \leftarrow (y_0 \ll i, y_1 \ll i)$,	# Shared SHIFT
2: $(s_0, s_1, s_2, s_3) \leftarrow (x_0 \wedge v_0, x_0 \wedge v_1, x_1 \wedge v_0, x_1 \wedge v_1)$	# Shared AND
3: $(t_0, t_1) \leftarrow (s_0 \oplus y_0, s_1 \oplus y_1)$	# Shared XOR
4: $(z_0, z_1) \leftarrow (t_0 \oplus s_2, t_1 \oplus s_3)$	# Collapse shares
5: **return** (z_0, z_1)	

In addition, we are interested in an integration of the shift operation into `SecAnd` and `SecAndXor` to be able to use the flexible second operand feature of the ARM instruction set. This operation takes only one shared input (x_0, x_1) and an additional shift parameter i. Then, the shared input (x_0, x_1) is shifted by i bits to the left to generate the second operand of `SecAnd`. This leads to the two shared operations `SecAndShift` (Algorithm 5) and `SecAndShiftXor` (Algorithm 6). Proving the uniformity of `SecAndShift` is essentially the same as Lemma 3, therefore we skip the proof of Lemma 4, because by construction, precondition 3 of Lemma 3 is automatically fullfilled, if (x_0, x_1) is a uniform sharing of x.

Lemma 4. *Algorithm 5 implements a correct, non-complete and uniform sharing of a k-radix AND-SHIFT gate with k inputs and k outputs:*

1. *if the inputs x_0, x_1 is a uniform sharing of x,*
2. *if u is uniformly distributed,*
3. *if the distributions of $x_0[k-1], x_1[k-1], x_0[k-i], x_1[k-i]$ are independent of the distribution of u,*
4. *and if the intermediate state (t_0, t_1) is stored to a register before computing the output sharing (z_0, z_1).*

The next important observation is that the least significant bit $x_0[0]$ is not used to guard any other output shares. Thus, this bit can be reused as guard share for the next series of parallel ANDs in the for loop of the KSA algorithm, which again leaves us with a single bit which we can reuse as guard share in the next iteration. We inductively prove this property by isolating the repeated AND-SHIFT gate which computes all the propagates in the for loop of the KSA algorithm (Algorithm 7).

Lemma 5. *Algorithm 7 implements a correct, non-complete and uniform sharing of the function $p[k-1] \leftarrow (x[0] \oplus y[0]) \wedge \cdots \wedge (x[k-1] \oplus y[k-1])$,*

- *if x_0, x_1, y_0, y_1 are uniform sharings of x, y,*
- *and if u is uniformly distributed and if its distribution is independent of the inputs $x_0[k-1], x_1[k-1], y_0[k-1], y_1[k-1]$.*

Algorithm 7 2-Share TI of the computation of propagates

Require: $x_0, x_1, y_0, y_1 \in \mathbb{Z}_{2^k}$, $k > 0$, $u \in \{0,1\}$, with $x = x_0 \oplus x_1$ and $y = y_0 \oplus y_1$.
Ensure: $p[k-1] = (x[0] \oplus y[0]) \wedge \cdots \wedge (x[k-1] \oplus y[k-1])$, with $p = p_0 \oplus p_1$
1: $n \leftarrow \max(\lceil \log_2(k-1) \rceil, 1)$
2: $(p_0, p_1) \leftarrow \texttt{SecXor}(x_0, x_1, y_0, y_1)$ # Shared XOR
3: **for** $i = 1$ to $n-1$ **do**
4: $v \leftarrow p_0[0]$ # Save guard share $i+1$
5: $(p_0, p_1) \leftarrow \texttt{SecAndShift}(p_0, p_1, u, 2^{i-1})$ # Shared AND-SHIFT
6: $u \leftarrow v$ # Use new guard share
7: **end for**
8: **return** (p_0, p_1)

Proof. The initial part (Line 2) before the for loop is correct, non-complete and uniform, since it is only a linear combination of uniformly distributed random inputs.

In the for loop (Lines 4-6) we first assume, that in iteration i, u is uniformly distributed and independent of the shares $p_0[k-1]$ and $p_1[k-1]$. Then, by Lemma 4, the output of the `SecAndShift` (p'_0, p'_1) is a correct, non-complete and uniform sharing. For iteration $i+1$, the new u' has to be uniformly distributed and independent of the outputs $p'_0[k-1]$ and $p'_1[k-1]$ of iteration i. The uniformity of the distribution of u' is derived from the uniformity of p_0, namely $u' = p_0[0]$. Furthermore, the independence of the distribution of u' from $p'_0[k-1]$ and $p'_1[k-1]$ follows, because the sharing $(p'_0[k-1], p'_1[k-1])$ has been refreshed using another independently uniformly distributed guard share, which is either u for index $k-1$ or $p_0[k-1-j]$ for $0 \leq j < k-2$.

Since the initial values for the loop iteration $i = 1$ are generated by a correct, non-complete and uniform sharing, the loop invariant, that u is uniformly distributed and independent of p_0, and p_1, holds. Therefore, Algorithm 7 implements a correct, non-complete and uniform sharing. □

The proof of Lemma 5 is a variant of Daemen's proof in [Dae17]. Besides the obvious similarity, the proof itself is different, because Daemen's proof only works for permutations and hence, it is not applicable to our setting of the modular addition. Instead of basing our proof on the property of a permutation, we show that the inputs to the shared AND

Algorithm 8 Kogge-Stone 2-Share Addition

Require: $x_0, x_1, y_0, y_1 \in \mathbb{Z}_{2^k}$, $k > 0$, $u \in \{0,1\}$, with $x = x_0 \oplus x_1$ and $y = y_0 \oplus y_1$
Ensure: $z = (x + y) \bmod 2^k$, with $z = z_0 \oplus z_1$
1: $n \leftarrow \max(\lceil \log_2(k-1) \rceil, 1)$
2: $v \leftarrow x_0 \bmod 2$ # Save next guard share
3: $(g_0, g_1) \leftarrow \texttt{SecAnd}(x_0, x_1, y_0, y_1, u)$ # Shared AND
4: $(p_0, p_1) \leftarrow \texttt{SecXor}(x_0, x_1, y_0, y_1)$ # Shared XOR
5: $u \leftarrow v$ # Update guard share
6: **for** $i = 1$ to $n-1$ **do**
7: $v \leftarrow p_0 \bmod 2$ # Save next guard share
8: $(g_0, g_1) \leftarrow \texttt{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{i-1})$ # Shared AND-SHIFT-XOR
9: $(p_0, p_1) \leftarrow \texttt{SecAndShift}(p_0, p_1, u, 2^{i-1})$ # Shared AND-SHIFT
10: $u \leftarrow v$ # Update guard share
11: **end for**
12: $(g_0, g_1) \leftarrow \texttt{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{n-1})$ # Shared AND-SHIFT-XOR
13: $(z_0, z_1) \leftarrow (x_0 \oplus y_0 \oplus 2g_0, x_1 \oplus y_1 \oplus 2g_1)$ # Compute final output
14: **return** (z_0, z_1, u)

gates and the guard masks are independently distributed in all cases during the for loop. Therefore, we can also show the uniformity property is fulfilled.

Using Lemma 5, we can now construct the 2-share TI of the full Kogge-Stone adder as depicted in Algorithm 8. The uniformity follows in a straight forward manner from Lemma 3, Lemma 5 and from a parallel version of Lemma 2. Therefore, we give the theorem without proof.

Theorem 1. *Algorithm 8 implements a correct, non-complete and uniform sharing of the Kogge-Stone Adder.*

In Appendix B, we also provide the subtraction variant (Algorithm 14) by making the same changes as we did to Biryukov et al.'s masked addition (Algorithm 13).

3.4 Extension to ARX ciphers

Typical state of the art ciphers use more than one addition operation. Therefore, it is an interesting additional question, if the remaining guard share from the adder can be used as input guard share for other additions. We show that this idea is indeed sound for the application in any ARX cipher. Therefore, we can implement any ARX cipher with either our first unoptimized 2-share KSA or with its optimized variant.

Theorem 2. *Reusing the unused guard share u from the last iteration in Algorithm 8 as guard share in a subsequent addition results in a correct, non-complete and uniform sharing of the subsequent addition in ChaCha20 (and all similar ARX ciphers).*

Proof. All outputs of the 2-share additions are reshared with inputs independent of the unused guard share u. Therefore, all outputs are independently distributed from u. Furthermore, all other shares of the internal state of the cipher are independently distributed and therefore, it is impossible that the distribution of u is dependent on any other bit of a sharing of the internal state of the cipher. □

3.5 Improved 2-Share TI Addition

Our new algorithm can be improved further by optimizing the shared AND (Algorithm 9), described in [BDLCU17]. The main benefit is a further reduction on the number of instructions. We like to highlight that the addition formula from Biryukov et al. could also be interpreted as 2-share TI that could be found by direct sharing and application of correction terms as proposed in the TI literature [BNN+12]. The equivalent formulas with correction terms in sum-of-products normal form are:

$$s_0 \leftarrow x_0 \wedge y_0, \qquad s_1 \leftarrow x_0 \wedge y_1 \oplus y_1 \oplus 1$$
$$s_2 \leftarrow x_1 \wedge y_0, \qquad s_3 \leftarrow x_1 \wedge y_1 \oplus y_1 \oplus 1$$
$$z_0 \leftarrow s_0 \oplus s_1, \qquad z_1 \leftarrow s_2 \oplus s_3$$

Therefore, a version which also integrates the shift operation (Algorithm 10) can be used as a drop-in replacement for our algorithm `SecAndShift`, which results in Algorithm 11.

Algorithm 9 Optimized 2-Share TI of a k-radix AND gate (`SecAndOpt`) [BDLCU17].

Require: $x_0, x_1, y_0, y_1 \in \mathbb{Z}_{2^k}$, $k > 0$, $u \in \{0, 1\}$, $0 \leq i < k$
Ensure: $(z_0 \oplus z_1) = (x_0 \oplus x_1) \wedge ((x_0 \oplus x_1) \ll i) \bmod 2^k$
1: $s_0 \leftarrow x_0 \wedge y_0$, $s_1 \leftarrow x_0 \vee \neg y_1$ # Shared AND
2: $s_2 \leftarrow x_1 \wedge y_0$, $s_3 \leftarrow x_1 \vee \neg y_1$
3: $z_0 \leftarrow s_0 \oplus s_1$, $z_1 \leftarrow s_2 \oplus s_3$
4: **return** (z_0, z_1)

Algorithm 10 Optimized 2-Share TI of a k-radix AND-SHIFT gate (`SecAndShiftOpt`) [BDLCU17].

Require: $x_0, x_1 \in \mathbb{Z}_{2^k}$, $k > 0$, $u \in \{0,1\}$, $0 \leq i < k$
Ensure: $(z_0 \oplus z_1) = (x_0 \oplus x_1) \wedge ((x_0 \oplus x_1) \ll i) \bmod 2^k$
1: $(y_0, y_1) \leftarrow (x_0 \ll i, x_1 \ll i)$ # Shared SHIFT
2: $(s_0, s_1, s_2, s_3) \leftarrow (x_0 \wedge y_0, x_0 \vee \neg y_1, x_1 \wedge y_0, x_1 \vee \neg y_1)$ # Shared AND
3: $(z_0, z_1) \leftarrow (s_0 \oplus s_1, s_2 \oplus s_3)$ # Collapse shares
4: **return** (z_0, z_1)

Algorithm 11 Kogge-Stone Optimized 2-Share Addition

Require: $x_1, x_2, y_1, y_2 \in \mathbb{Z}_{2^k}$, $k > 0$, $u \in \{0,1\}$, with $x = x_1 \oplus x_2$ and $y = y_1 \oplus y_2$
Ensure: $z = (x + y) \bmod 2^k$, with $z = z_1 \oplus z_2$
1: $n \leftarrow \max(\lceil \log_2(k-1) \rceil, 1)$
2: $m \leftarrow ((x_0 \gg 1) \oplus (u \ll (k-1))$ # Generate refresh mask
3: $(g_0, g_1) \leftarrow$ `SecAndOpt`(x_0, x_1, y_0, y_1) # Shared AND
4: $(p_0, p_1) \leftarrow$ `SecXor`(x_0, x_1, y_0, y_1) # Shared XOR
5: $(g_0, g_1) \leftarrow (g_0 \oplus m, g_1 \oplus m)$ # Refresh sharing of g
6: $u \leftarrow x_0[0]$ # Update guard share
7: **for** $i = 1$ to $n - 1$ **do**
8: $\quad (g_0, g_1) \leftarrow$ `SecAndShiftXor`$(p_0, p_1, g_0, g_1, 2^{i-1})$ # Shared AND-SHIFT-XOR
9: $\quad (p_0, p_1) \leftarrow$ `SecAndShiftOpt`$(p_0, p_1, 2^{i-1})$ # Shared AND-SHIFT
10: **end for**
11: $(g_0, g_1) \leftarrow$ `SecAndShiftXor`$(p_0, p_1, g_0, g_1, 2^{n-1})$ # Shared AND-SHIFT-XOR
12: $(z_0, z_1) \leftarrow (x_0 \oplus y_0 \oplus 2g_0, x_1 \oplus y_1 \oplus 2g_1)$
13: **return** (z_0, z_1, u)

For `SecAndShiftXor` there is no performance benefit, because the number of instructions will be worse in a generic setting, if the optimized masked AND gate is combined with the shared XOR gate and identical for ARM.

Regarding the additional entropy, it is important to note, that the composition in Algorithm 11 would not work without the additional refreshing of randomness before the loop. This problem arises, because the outputs of the `SecAndOpt` and `SecXor` gates are not statistically independent and hence, we would observe leakage during the computation of the shared generates. The refreshing makes the distributions of the sharing of g and the sharing of p independent of each other before the loop. This independence is then preserved because the `SecAndShiftXor` gate uses the sharing (g_0, g_1) to refresh the entropy and since the initial distributions of (g_0, g_1) and (p_0, p_1) in the loop are independent of each other, the independence is always preserved throughout the loop. In the following theorem, we show that the 2-Share TI KSA shown in Algorithm 11 is a proper 2-share TI which needs only one refreshing and hence, a one bit guard share u.

Theorem 3. *Algorithm 11 implements a correct, non-complete and uniform sharing of the Kogge-Stone Adder.*

Proof. The correctness and non-completeness follows from the usage of the components `SecXor`, `SecAndOpt`, `SecAndShiftXor`, and `SecAndShiftOpt`. The mask refresh on Line 5 also does not change the correctness of the implementation and is obviously non-complete.

For the uniformity, we investigate the individual components:

1. `SecAndOpt` (Line 3) and `SecXor` (Line 4) produce uniform, but not independent sharings.

2. The mask refresh on Line 5 restores the independence of the shared propagates (p_0, p_1) which are produced by the `SecAndOpt` operation (Line 3) from the generates (g_0, g_1).

3. For the uniformity of the loop, we first show, that a simplified variant of Lemma 5 also holds for `SecAndShiftOpt`. The sharing of the AND-SHIFT gate in the for loop (Line 7 in Algorithm 11) is a uniform sharing, if the input is uniformly shared. The uniformity condition is sufficient, because the shift operation guarantees the independence of the two shared inputs to the original masked AND gate. Therefore, for iteration i, the output of the AND-SHIFT gate is a uniform sharing of the original AND-SHIFT output. Furthermore, the input to the AND-SHIFT gate for iteration $i+1$ is also a uniform sharing, since it is the output of iteration i. Therefore, also the output for iteration $i+1$ is a uniform sharing. Since the initial sharing (p_0, p_1) for iteration 1 is also uniform, the AND-SHIFT gate in the for loop (Line 7 in Algorithm 11) only produces uniform sharings of the intermediates of the propagates.

4. For the AND-SHIFT-XOR operation in the loop, we can perform a similar proof than for the AND-SHIFT gate. Firstly, we assume that (p_0, p_1) and (g_0, g_1) are independently uniformly distributed sharings in iteration i. Then the output of AND-SHIFT-XOR is also a uniform sharing which is independently distributed of the sharing (p_0, p_1). Secondly, since the output of AND-SHIFT-XOR in iteration i is uniform, the inputs g_0, g_1 in iteration $i+1$ are uniform. Furthermore, we already proofed that in all loop iterations (p_0, p_1) is a uniform sharing of p. Therefore, also the output in iteration $i+1$ is uniform. Since the inputs in iteration 1 are also uniform and independently distributed, all output sharings during the loop computation are also uniform.

5. The uniformity of the AND-SHIFT-XOR gate on line 11 follows, because the outputs of the loop are independently uniformly distributed, and they are the inputs of this operation.

6. On line 12, all operations are linear combinations of uniform sharings and are therefore obviously uniform.

Since all operations have uniform input and output sharings, the optimized KSA is also uniform and hence a correct, non-complete and uniform 2-share TI. □

3.6 Operation Count

In Table 1, we show the detailed operation counts for the individual operations. The generic architecture assumes, that there are only four operations available, NOT, AND, XOR and SHIFT, while we allow the full ARM instruction set for the ARM figures. The table shows, that our new operations are competitive in general and especially, that merging operations can lead to considerable performance savings, without compromising the theoretical security, even though the formulas presented in [BDLCU17] are optimal for the two-input gates. Although our `SecAnd` operation is slower in the generic setting than the variants from [CGTV15, BDLCU17], this is mainly due to the integration of the mask generation. If we disregard this and if we look a the ARM results for `SecAndShift`, the number of instructions for ARM does not increase, since the shift can be integrated in the same instruction and hence, the operation is faster. Furthermore, if we use the optimizations proposed by Biryukov et al., we can further reduce the instruction count for the ARM implementation of `SecAndShift` to only 6 instructions, which saves 25% instructions on this important platform. In addition, we note, that using the `SecAndShiftXor` seems to be beneficial in all settings, which is due to merging AND, SHIFT and XOR in a single operation.

Table 1: Comparison of masked operations.

	SecXor	SecShift	SecAnd	SecAndShift/-Opt	SecAndShiftXor
Generic [CGTV15]	2	4	8	$8+2$	$8+4+2$
ARM [CGTV15]	2	4	8	$8+2$	$8+4+2$
Generic [BDLCU17]	2	2	7	$7+2$	$7+2+2$
ARM [BDLCU17]	2	2	6	$6+2$	$6+2+2$
Generic [new]	2	-	8 (11)[1]	10 (13)[1]/9	10
ARM [new]	2	-	8 (11)[1]	8 (11)[1]/6	8

[1] Value in parenthesis includes generation of refresh mask.

Table 2: Comparison of masked addition and subtraction.

	Addition	Opt. Addition	Subtraction	Opt. Subtraction	Rand.
Generic [CGTV15]	$28\log_2 k+4$	-	-	-	k
ARM [CGTV15]	$28\log_2 k+4$	-	-	-	k
Generic [BDLCU17]	$22\log_2 k+6$	-	$32\log_2 k+4$	-	0
ARM [BDLCU17]	$22\log_2 k+4$	-	$30\log_2 k+6$	-	0
Generic [new]	$24\log_2 k+6$	$19\log_2 k+11$	$24\log_2 k+12$	$19\log_2 k+17$	1
ARM [new]	$18\log_2 k+4$	$14\log_2 k+8$	$18\log_2 k+11$	$14\log_2 k+13$	1

A similar effect can be seen in Table 2, where we see that the removal of some mask refreshing steps, the usage of the second operand feature and also other optimizations on the assembly level can outperform the previously published results significantly. Our basic addition variant already improves the state of the art for ARM implementations about 18%, but when we also take the other optimizations by Biryukov et al. into account, we end up with an even more impressive saving of about 50% compared to the results from [CGTV15] and 36% compared to [BDLCU17]. Furthermore, the improved subtraction algorithm is about 53% faster than the one proposed in [BDLCU17]. Overall, for one of the most common values of k, $k = 32$, we reduce the overhead from 114 instructions [BDLCU17] to only 83. This is a significant cost saving, especially, if we consider that an unprotected addition usually takes only one clock cycle on most common platforms. As a reference, we also append ARM implementations of our two addition implementations in Appendix E. However, we only made sure that no basic distance-based leakage from register writes are existing in the implementations and hence, the implementations are likely to leak in practice and therefore, they only serve as a starting point for real world implementations.

4 Implementation Overview

For our evaluation, we developed several ARM assembly implementations of ChaCha20, targeting ARM Cortex-M3 and Cortex-M4 processors. We implemented an unprotected reference according to the recommendations reported in [SSS17], to serve as a baseline to compare the relative performance overhead of our protected implementations. Furthermore, we developed two protected implementations of ChaCha20, one with our first unoptimized masked addition and the second with the optimized counterpart. All of our performance numbers are reported in Table 3 alongside other implementation results from [AFM17, SSS17]. However, due to different hardware, the clock cycle counts are not fully comparable, because the memory architecture plays an important role, even if the basic ARM architecture is identical.

In most cases, we used the optimizations such as the second flexible operand, as described in [SSS17], to reduce the cycle count. Hence, our unprotected version performs

Table 3: Code sizes and runtime cycle counts for the tested implementations

Implementation	Code	Stack	Cycles
Our Reference [new]	488	56	1726
Unprotected [AFM17]	N/A	N/A	4380
Unprotected [SSS17]	734	232	1487
Unprotected [SSS17]	3174	228	1287
2-Share [new]	2214	316	72721
2-Share Optimized [new]	2138	316	60623
Masked [AFM17]	N/A	N/A	121618
Masked [AFM17]	N/A	N/A	93993

roughly as fast as reported in [SSS17].

The size and time overheads of the protected implementations in comparison to our reference are considerable. Both 2-Share variants exhibit a 4.5 fold increase in size, and a 35× to 42× increase in duration. This overhead is much higher than the 21× increase reported in [AFM17], however, our reference implementation is also much faster than theirs.

Note that the optimized 2-share variant is about 17% faster than our first implementation, a figure close to the 27% difference in the number of instructions presented in Table 2 (Section 3.6). Furthermore, we beat the results reported in [AFM17] by 36%. This shows, that the masked addition is the driving factor for the speed of the countermeasure. The assembly implementations for the masked additions are shown in Appendix E.

5 Conclusion

ARX ciphers such as ChaCha20 are deployed in many application domains. A major benefit of such ciphers is that they are easy to protect against timing side-channel attacks. However, other side-channel protections, such as against power or EM are less straightforward and all secure state of the art protections are very costly.

Our results promise a significant performance improvement for Boolean masking for the modular addition. With a reduction of the instruction count by 36% over the state of the art masked adder and more than 50% for masked subtraction, our proposed algorithms provide a major performance improvement. This also translates to a much faster implementation for ChaCha20 on Cortex-M3/M4 processors.

At the same time, we cut down the requirement of randomness to only one additional bit, which is very close to the randomness-free adder by Biryukov et al. [BDLCU17]. Therefore, only 513 bits of randomness have to be sampled per encryption (or decryption) in the case of ChaCha20. Overall, the performance improvements and the much lower randomness requirements are very helpful when ARX ciphers have to be protected against side-channel attacks at a low cost. However, despite our improvements, the penalty for masking ARX ciphers is still very high and hence, further research is necessary. One direction could be an investigation of bit-sliced implementations, especially for processors where SIMD operations are available.

It is also noteworthy, that we used the TI methodology with two shares to beat a conventional Boolean masking implementation. To the best of our knowledge this is the first report, that TI can also deliver competitive performance for software implementations.

References

[AFM17] Alexandre Adomnicai, Jacques J A Fournier, and Laurent Masson. Bricklayer Attack: A Side-Channel Analysis on the ChaCha Quarter Round. In *Progress in Cryptology – INDOCRYPT '17*, pages 65–84. Springer-Verlag, 2017.

[BDF+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model. In *Advances in Cryptology – EUROCRYPT '17*, pages 535–566. Springer-Verlag, 2017.

[BDLCU17] Alex Biryukov, Daniel Dinu, Yann Le Corre, and Aleksei Udovenko. Optimal First-Order Boolean Masking for Embedded IoT Devices. In *International Conference on Smart Card Research and Advanced Applications – CARDIS '17*, pages 22–41. Springer-Verlag, 2017.

[Ber08] Daniel J Bernstein. ChaCha, a variant of Salsa20. In *The State of the Art of Stream Ciphers – SASC '08*. ECRYPT, 2008.

[BGG+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the Cost of Lazy Engineering for Masked Software Implementations. In *International Conference on Smart Card Research and Advanced Applications – CARDIS '14*, pages 64–81. Springer-Verlag, 2014.

[BGRV15] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. DPA, Bitslicing and Masking at 1 GHz. In *International Workshop on Cryptographic Hardware and Embedded Systems – CHES '15*, pages 599–619. Springer-Verlag, 2015.

[BNN+12] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, and Georg Stütz. Threshold Implementations of All 3×3 and 4×4 S-Boxes. In *International Workshop on Cryptographic Hardware and Embedded Systems – CHES '12*, pages 76–91. Springer-Verlag, 2012.

[BSS+15] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *52nd Design Automation Conference – DAC '15*, pages 1–6. ACM/IEEE, 2015.

[CFE16] Cong Chen, Mohammad Farmani, and Thomas Eisenbarth. A Tale of Two Shares: Why Two-Share Threshold Implementation Seems Worthwhile—and Why It Is Not. In *Advances in Cryptology – ASIACRYPT '16*, pages 819–843. Springer-Verlag, 2016.

[CG00] Jean-Sébastien Coron and Louis Goubin. On Boolean and Arithmetic Masking against Differential Power Analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems – CHES '00*, pages 231–237. Springer-Verlag, 2000.

[CGD17] Yann Le Corre, Johann Großschädl, and Daniel Dinu. Micro-Architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors. Cryptology ePrint Archive, Report 2017/1253, 2017. https://eprint.iacr.org/2017/1253.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. In *International Workshop on Fast Software Encryption – FSE '15*, pages 130–149. Springer-Verlag, 2015.

[Dae17] Joan Daemen. Changing of the Guards: A Simple and Efficient Method for Achieving Uniformity in Threshold Sharing. In *International Conference on Cryptographic Hardware and Embedded Systems – CHES '17*, pages 137–153. Springer-Verlag, 2017.

[DCBG+17] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. Does Coupling Affect the Security of Masked Implementations? In *International Workshop on Constructive Side-Channel Analysis and Secure Design – COSADE '17*, pages 1–18, Cham, 2017. Springer-Verlag.

[DCRB+16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with $d+1$ Shares in Hardware. In *International Conference on Cryptographic Hardware and Embedded Systems – CHES '16*, pages 194–212. Springer-Verlag, 2016.

[DGLC17] Daniel Dinu, Johann Großschädl, and Yann Le Corre. Efficient Masking of ARX-Based Block Ciphers Using Carry-Save Addition on Boolean Shares. In *International Conference on Information Security – ISC '17*, pages 39–57. Springer-Verlag, 2017.

[FLS+10] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein Hash Function Family. Technical report, 2010.

[GMK16] Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. Cryptology ePrint Archive, Report 2016/486, 2016. https://eprint.iacr.org/2016/486.

[Gou01] Louis Goubin. A Sound Method for Switching between Boolean and Arithmetic Masking. In *International Workshop on Cryptographic Hardware and Embedded Systems – CHES '01*, pages 3–15. Springer-Verlag, 2001.

[KNPW13] Sebastian Kutzner, Phuong Ha Nguyen, Axel Poschmann, and Huaxiong Wang. On 3-Share Threshold Implementations for 4-Bit S-boxes. In *International Workshop on Constructive Side-Channel Analysis and Secure Design – COSADE '13*, pages 99–113. Springer-Verlag, 2013.

[KS73] Peter M Kogge and Harold S Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22(8):786–793, 1973.

[LF80] Richard E Ladner and Michael J Fischer. Parallel Prefix Computation. *Journal of the ACM – JACM*, 27(4):831–838, 1980.

[Ltd18] ARM Ltd. Arm® v8-M Architecture Reference Manual, 2018.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer-Verlag, 2007.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In *International Conference on Information and Communications Security – ICICS '06*, pages 529–545. Springer-Verlag, 2006.

[NRS08] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure Hardware Implementation of Non-linear Functions in the Presence of Glitches. In *International Conference on Information Security and Cryptology – ICISC '08*, pages 218–234. Springer-Verlag, 2008.

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the Gap: Towards Secure 1st-Order Masking in Software. In *International Workshop on Constructive Side-Channel Analysis and Secure Design – COSADE '17*, pages 282–297. Springer-Verlag, 2017.

[RBN+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating Masking Schemes. In *Advances in Cryptology – CRYPTO '15*, pages 764–783. Springer-Verlag, 2015.

[RL13] Thomas Roche and Victor Lomné. Collision-Correlation Attack against Some 1st-Order Boolean Masking Schemes in the Context of Secure Devices. In *International Workshop on Constructive Side-Channel Analysis and Secure Design – COSADE '13*, pages 114–136. Springer-Verlag, 2013.

[Ros60] Gerald B. Rosenberger. Simultaneous Carry Adder, 1960. US Patent 2,966,305.

[SMG15] Tobias Schneider, Amir Moradi, and Tim Güneysu. Arithmetic Addition over Boolean Masking. In *International Conference on Applied Cryptography and Network Security – ACNS '15*, pages 559–578. Springer-Verlag, 2015.

[SS16] Peter Schwabe and Ko Stoffelen. All the AES You Need on Cortex-M3 and M4. In *International Conference on Selected Areas in Cryptography – SAC '16*, pages 180–194. Springer-Verlag, 2016.

[SSS17] Fabrizio De Santis, Andreas Schauer, and Georg Sigl. ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications. In *Design, Automation, Test in Europe Conference – DATE '17*, pages 692–697. IEEE, 2017.

[STM14] STMicroelectronics. STM32F411xC/E advanced ARM®-based 32-bit MCUs – Reference Manual – RM0383. Online Documentation, 2014. http://www.st.com/resource/zh/reference_manual/dm00119316.pdf.

[WH17] Yoo-Seung Won and Dong-Guk Han. Efficient Conversion Method from Arithmetic to Boolean Masking in Constrained Devices. In *International Workshop on Constructive Side-Channel Analysis and Secure Design – COSADE '17*, pages 120–137. Springer-Verlag, 2017.

[ZKS12] Michael Zohner, Michael Kasper, and Marc Stöttinger. Butterfly-Attack on Skein's Modular Addition. In *International Workshop on Constructive Side-Channel Analysis and Secure Design – COSADE '12*, pages 215–230. Springer-Verlag, 2012.

A Algorithms for Optimized Boolean Masking [BDLCU17]

This appendix contains the original masked adder (Algorithm 12) using the optimized formula from Biryukov et al. [BDLCU17]. We also show our optimized subtraction operation based on the same basic ideas (Algorithm 13).

Algorithm 12 Kogge-Stone Masked Addition [BDLCU17]

Require: $x_0, x_1, y_0, y_1 \in \mathbb{Z}_{2^k}$, with $x = x_0 \oplus x_1$ and $y = y_0 \oplus y_1$
Ensure: $z = (x + y) \bmod 2^k$, with $z = z_0 \oplus z_1$
1: $n \leftarrow \max(\lceil \log_2(k-1) \rceil, 1)$
2: $(p_0, p_1) \leftarrow \texttt{SecXor}(x_0, x_1, y_0, y_1)$
3: $(g_0, g_1) \leftarrow \texttt{SecAnd}(x_0, x_1, y_0, y_1)$
4: $(g_0, g_1) \leftarrow ((g_0 \oplus x_1) \oplus g_1, x_1)$
5: **for** $i = 1$ to $n - 1$ **do**
6: $\quad(h_0, h_1) \leftarrow \texttt{SecShift}(g_0, g_1, 2^{i-1})$
7: $\quad(u_0, u_1) \leftarrow \texttt{SecAnd}(p_0, p_1, h_0, h_1)$
8: $\quad(g_0, g_1) \leftarrow \texttt{SecXor}(g_0, g_1, u_0, u_1)$
9: $\quad(h_0, h_1) \leftarrow \texttt{SecShift}(p_0, p_1, 2^{i-1})$
10: $\quad(h_0, h_1) \leftarrow ((h_0 \oplus x_1) \oplus h_1, x_1)$
11: $\quad(p_0, p_1) \leftarrow \texttt{SecAnd}(p_1, p_2, h_0, h_1)$
12: $\quad(p_0, p_1) \leftarrow ((p_0 \oplus y_1) \oplus p_1, y_1)$
13: **end for**
14: $(h_0, h_1) \leftarrow \texttt{SecShift}(g_0, g_1, 2^{n-1})$
15: $(u_0, u_1) \leftarrow \texttt{SecAnd}(p_0, p_1, h_0, h_1)$
16: $(g_0, g_1) \leftarrow \texttt{SecXor}(g_0, g_1, u_0, u_1)$
17: $(z_0, z_1) \leftarrow \texttt{SecXor}(y_0, y_1, x_0, x_1)$
18: $(p_0, p_1) \leftarrow ((z_0 \oplus (g_0 \ll 1)) \oplus (x_1 \ll 1), y_1)$
19: **return** (z_0, z_1)

Algorithm 13 Kogge-Stone Masked Subtraction (based on adder from [BDLCU17])

Require: $x_1, x_2, y_1, y_2 \in \mathbb{Z}_{2^k}$, with $x = x_1 \oplus x_2$ and $y = y_1 \oplus y_2$
Ensure: $z = (x - y) \bmod 2^k$, with $z = z_1 \oplus z_2$
1: $n \leftarrow \max(\lceil \log_2(k-1) \rceil, 1)$
2: $\boldsymbol{y_1 \leftarrow \neg y_1}$
3: $(p_0, p_1) \leftarrow \texttt{SecXor}(x_0, x_1, y_0, y_1)$
4: $(g_0, g_1) \leftarrow \texttt{SecAnd}(x_0, x_1, y_0, y_1)$
5: $(g_0, g_1) \leftarrow ((g_0 \oplus x_1) \oplus g_1, x_1)$
6: $\boldsymbol{(g_0, g_1) \leftarrow ((g_0 \oplus (y_0 \wedge 1)) \oplus (y_1 \wedge 1), g_1)}$
7: **for** $i = 1$ to $n - 1$ **do**
8: $\quad (h_0, h_1) \leftarrow \texttt{SecShift}(g_0, g_1, 2^{i-1})$
9: $\quad (u_0, u_1) \leftarrow \texttt{SecAnd}(p_0, p_1, h_0, h_1)$
10: $\quad (g_0, g_1) \leftarrow \texttt{SecXor}(g_0, g_1, u_0, u_1)$
11: $\quad (h_0, h_1) \leftarrow \texttt{SecShift}(p_0, p_1, 2^{i-1})$
12: $\quad (h_0, h_1) \leftarrow ((h_0 \oplus x_1) \oplus h_1, x_1)$
13: $\quad (p_0, p_1) \leftarrow \texttt{SecAnd}(p_1, p_2, h_0, h_1)$
14: $\quad (p_0, p_1) \leftarrow ((p_0 \oplus y_1) \oplus p_1, y_1)$
15: **end for**
16: $(h_0, h_1) \leftarrow \texttt{SecShift}(g_0, g_1, 2^{n-1})$
17: $(u_0, u_1) \leftarrow \texttt{SecAnd}(p_0, p_1, h_0, h_1)$
18: $(g_0, g_1) \leftarrow \texttt{SecXor}(g_0, g_1, u_0, u_1)$
19: $(z_0, z_1) \leftarrow \texttt{SecXor}(y_0, y_1, x_0, x_1)$
20: $(z_0, z_1) \leftarrow ((z_0 \oplus (g_0 \ll 1)) \oplus (x_1 \ll 1), y_1)$
21: $\boldsymbol{(z_0, z_1) \leftarrow (z_0 \oplus 1, z_1)}$
22: **return** (z_0, z_1)

B Algorithms for 2-Share Subtraction

This appendix contains the 2-share subtraction variant of our 2-share addition operations, both the first variant (Algorithm 14) and also the version optimized with Biryukov et al.'s AND formula (Algorithm 15).

Algorithm 14 Kogge-Stone 2-Share Subtraction

Require: $x_1, x_2, y_1, y_2 \in \mathbb{Z}_{2^k}$, $k > 0$, $u \in \{0, 1\}$, with $x = x_1 \oplus x_2$ and $y = y_1 \oplus y_2$
Ensure: $z = (x - y) \bmod 2^k$, with $z = z_1 \oplus z_2$

1: $n \leftarrow \max(\lceil \log_2(k-1) \rceil, 1)$
2: $v \leftarrow x_0 \bmod 2$ # Save next guard share
3: $\boldsymbol{y_1 \leftarrow \neg y_1}$
4: $(g_0, g_1) \leftarrow \texttt{SecAnd}(x_0, x_1, y_0, y_1, u)$ # Shared AND
5: $(p_0, p_1) \leftarrow \texttt{SecXor}(x_0, x_1, y_0, y_1)$ # Shared XOR
6: $\boldsymbol{(g_1, g_2) \leftarrow (g_1 \oplus (y_1 \wedge 1), g_2 \oplus (y_2 \wedge 1))}$
7: $u \leftarrow v$ # Update guard share
8: **for** $i = 1$ to $n - 1$ **do**
9: $v \leftarrow p_0 \bmod 2$ # Save next guard share
10: $(g_0, g_1) \leftarrow \texttt{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{i-1})$ # Shared AND-SHIFT-XOR
11: $(p_0, p_1) \leftarrow \texttt{SecAndShift}(p_0, p_1, u, 2^{i-1})$ # Shared AND-SHIFT
12: $u \leftarrow v$ # Update guard share
13: **end for**
14: $(g_0, g_1) \leftarrow \texttt{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{n-1})$ # Shared AND-SHIFT-XOR
15: $(z_0, z_1) \leftarrow (x_0 \oplus y_0 \oplus 2g_0, x_1 \oplus y_1 \oplus 2g_1)$ # Compute final output
16: $\boldsymbol{(z_1, z_2) \leftarrow (z_1 \oplus 1, z_2)}$
17: **return** (z_0, z_1, u)

Algorithm 15 Kogge-Stone Optimized 2-Share Subtraction

Require: $x_1, x_2, y_1, y_2 \in \mathbb{Z}_{2^k}$, $k > 0$, $u \in \{0, 1\}$, with $x = x_1 \oplus x_2$ and $y = y_1 \oplus y_2$
Ensure: $z = (x - y) \bmod 2^k$, with $z = z_1 \oplus z_2$

1: $n \leftarrow \max(\lceil \log_2(k-1) \rceil, 1)$
2: $m \leftarrow ((x_0 \gg 1) \oplus (u \ll (k-1))$ # Generate refresh mask
3: $\boldsymbol{y_1 \leftarrow \neg y_1}$
4: $(g_0, g_1) \leftarrow \texttt{SecAndOpt}(x_0, x_1, y_0, y_1)$ # Shared AND
5: $(p_0, p_1) \leftarrow \texttt{SecXor}(x_0, x_1, y_0, y_1)$ # Shared XOR
6: $(g_0, g_1) \leftarrow (g_0 \oplus m, g_1 \oplus m)$ # Refresh sharing of g
7: $\boldsymbol{(g_1, g_2) \leftarrow (g_1 \oplus (y_1 \wedge 1), g_2 \oplus (y_2 \wedge 1))}$
8: $u \leftarrow x_0[0]$ # Update guard share
9: **for** $i = 1$ to $n - 1$ **do**
10: $(g_0, g_1) \leftarrow \texttt{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{i-1})$ # Shared AND-SHIFT-XOR
11: $(p_0, p_1) \leftarrow \texttt{SecAndShiftOpt}(p_0, p_1, 2^{i-1})$ # Shared AND-SHIFT
12: **end for**
13: $(g_0, g_1) \leftarrow \texttt{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{n-1})$ # Shared AND-SHIFT-XOR
14: $(z_0, z_1) \leftarrow (x_0 \oplus y_0 \oplus 2g_0, x_1 \oplus y_1 \oplus 2g_1)$ # Compute final output
15: $\boldsymbol{(z_1, z_2) \leftarrow (z_1 \oplus 1, z_2)}$
16: **return** (z_0, z_1, u)

C Proof of Uniformity for Lemma 1

Listing 1 Proof of Uniformity for Lemma 1

```
import numpy as np

distribution = np.zeros(4)

for x in xrange(0,4):
  x0 = x >> 1
  x1 = x & 1

  for y in xrange(0,4):
    y0 = y >> 1
    y1 = y & 1

    for m in xrange(0,2):
      s0 = x0 & y0
      s1 = x1 & y0
      s2 = x0 & y1
      s3 = x1 & y1

      t0 = s0 ^ m
      t1 = s1 ^ m

      z0 = t0 ^ s2
      z1 = t1 ^ s3

      distribution[z0 << 1 | z1] += 1

# distribution[3] and distribution[0] are representations of '0'
# Therefore, they must be equal to fulfill uniformity.
assert(distribution[3] == distribution[0])
# distribution[3] and distribution[0] are representations of '1'
# Therefore, they must be equal to fulfill uniformity.
assert(distribution[2] == distribution[1])
# The AND operation outputs three times '0' and once '1'.
assert(distribution[3]/distribution[2] == 3)

print "The sharing of the AND gate is uniform"
```

D Proof of Uniformity for Lemma 2

Listing 2 Proof of Uniformity for Lemma 2

```
import numpy as np

distribution = np.zeros(4)

for x in xrange(0,4):
  x0 = x >> 1
  x1 = x & 1

  for y in xrange(0,4):
    y0 = y >> 1
    y1 = y & 1

    for u in xrange(0,4):
      u0 = u >> 1
      u1 = u & 1

      s0 = x0 & y0
      s1 = x1 & y0
      s2 = x0 & y1
      s3 = x1 & y1

      t0 = s0 ^ u0
      t1 = s1 ^ u1

      z0 = t0 ^ s2
      z1 = t1 ^ s3

      distribution[z0 << 1 | z1] += 1

# distribution[3] and distribution[0] are representations of '0'
# Therefore, they must be equal to fulfill uniformity.
assert(distribution[3] == distribution[0])
# distribution[2] and distribution[1] are representations of '1'
# Therefore, they must be equal to fulfill uniformity.
assert(distribution[2] == distribution[1])
# The AND-XOR gate outputs '0' and '1' with the same frequency.
assert(distribution[3]/distribution[2] == 1)

print "The sharing of the AND-XOR gate is uniform"
```

E Addition ARM Assembly

Listing 3 and 4 show the assembly code for the 2-Share TI addition presented in this work. In both cases, the registers `r0` and `r1` contain the shares for the first operand, `r2` and `r3` the shares for the second. The resulting shares are placed in registers `r0` and `r1`. Register `r12` contains the additional 1-bit random value for mask refreshing.

Listing 3 2-Share TI Addition ARM implementation

```
// Generate mask
orr r4, r12, r0, lsr #1
lsl r12, r0, #31
and r5, r0, r2
and r6, r0, r3
eor r0, r0, r2
eor r5, r5, r4
eor r6, r6, r4
and r9, r1, r2
and r8, r1, r3
eor r1, r1, r3
eor r5, r5, r8
eor r6, r6, r9
// Iteration 1
// Generate mask
orr r4, r12, r0, lsr #1
lsl r12, r0, #31
and r7, r0, r5, lsl #1
and r8, r1, r5, lsl #1
and r10, r0, r6, lsl #1
and r11, r1, r6, lsl #1
eor r6, r6, r11
eor r6, r6, r10
eor r5, r7, r5
eor r5, r8, r5
and r2, r0, r0, lsl #1
and r3, r1, r0, lsl #1
and r10, r0, r1, lsl #1
and r11, r1, r1, lsl #1
eor r8, r11, r4
eor r8, r10, r8
eor r7, r3, r4
eor r7, r7, r2
// Iteration 2
// Generate mask
orr r4, r12, r7, lsr #1
lsl r12, r7, #31
and r2, r7, r5, lsl #2
and r3, r8, r5, lsl #2
and r10, r7, r6, lsl #2
and r11, r8, r6, lsl #2
eor r6, r6, r11
eor r6, r6, r10
eor r5, r2, r5
eor r5, r3, r5
and r2, r7, r7, lsl #2
and r3, r8, r7, lsl #2
and r10, r7, r8, lsl #2
and r11, r8, r8, lsl #2
eor r8, r11, r4
eor r8, r10, r8
eor r7, r3, r4
eor r7, r7, r2
// Iteration 3
// Generate mask
orr r4, r12, r7, lsr #1
lsl r12, r7, #31
and r2, r7, r5, lsl #4
and r3, r7, r6, lsl #4
and r9, r8, r5, lsl #4
and r10, r8, r6, lsl #4
eor r5, r5, r2
eor r6, r6, r3
eor r5, r5, r9
eor r6, r6, r10
and r2, r7, r7, lsl #4
and r3, r7, r8, lsl #4
and r9, r8, r7, lsl #4
and r10, r8, r8, lsl #4
eor r7, r2, r4
eor r8, r3, r4
eor r7, r7, r10
eor r8, r8, r9
// Iteration 4
// Generate mask
orr r4, r12, r7, lsr #1
lsl r12, r7, #31
and r2, r7, r5, lsl #8
and r3, r7, r6, lsl #8
and r9, r8, r5, lsl #8
and r10, r8, r6, lsl #8
eor r5, r5, r2
eor r6, r6, r3
eor r5, r5, r9
eor r6, r6, r10
and r2, r7, r7, lsl #8
and r3, r7, r8, lsl #8
and r9, r8, r7, lsl #8
and r10, r8, r8, lsl #8
eor r7, r2, r4
eor r8, r3, r4
eor r7, r7, r10
eor r8, r8, r9
// Post loop
and r2, r7, r5, lsl #16
and r3, r7, r6, lsl #16
and r9, r8, r5, lsl #16
and r10, r8, r6, lsl #16
eor r5, r5, r2
eor r6, r6, r3
eor r5, r5, r9
eor r6, r6, r10
eor r0, r0, r5, lsl #1
eor r1, r1, r6, lsl #1
```

The reference implementations given in this section are given to study the minimum number of assembly instructions for a masked 32 bit addition using our algorithms. If simulated with the MAPS simulator [CGD17] (without simulating the pipeline) the simulation confirms that there is no (distance-based) leakage. In a real world implementation, these implementations have to be adapted to remove other sources of leakage, such as pipeline leakages.

Listing 4 Optimized 2-Share TI Addition ARM implementation

```
1  // Generate mask
2  orr r11, r12, r0, lsr #1
3  lsl r12, r0, #31
4  orn r4, r0, r3
5  and r6, r2, r0
6  orn r5, r1, r3
7  and r7, r2, r1
8  eor r5, r7, r5
9  eor r4, r6, r4
10 eor r2, r2, r0
11 eor r3, r1, r3
12 eor r5, r5, r11
13 eor r4, r11, r4
14 // Iteration 1
15 and r8, r3, r4, lsl #1
16 and r9, r2, r4, lsl #1
17 eor r4, r9, r4
18 eor r4, r8, r4
19 and r10, r3, r5, lsl #1
20 and r11, r2, r5, lsl #1
21 eor r5, r10, r5
22 eor r5, r11, r5
23 orn r8, r3, r3, lsl #1
24 and r10, r3, r2, lsl #1
25 orn r9, r2, r3, lsl #1
26 and r11, r2, r2, lsl #1
27 eor r7, r10, r8
28 eor r6, r9, r11
29 // Iteration 2
30 and r8, r7, r4, lsl #2
31 and r9, r6, r4, lsl #2
32 eor r4, r9, r4
33 eor r4, r8, r4
34 and r10, r7, r5, lsl #2
35 and r11, r6, r5, lsl #2
36 eor r5, r10, r5
37 eor r5, r11, r5
38 orn r8, r7, r7, lsl #2
39 and r10, r7, r6, lsl #2
40 orn r9, r6, r7, lsl #2
41 and r11, r6, r6, lsl #2
42 eor r7, r10, r8
43 eor r6, r9, r11
43 // Iteration 3
44 and r8, r7, r4, lsl #4
45 and r9, r6, r4, lsl #4
46 eor r4, r9, r4
47 eor r4, r8, r4
48 and r10, r7, r5, lsl #4
49 and r11, r6, r5, lsl #4
50 eor r5, r10, r5
51 eor r5, r11, r5
52 orn r8, r7, r7, lsl #4
53 and r10, r7, r6, lsl #4
54 orn r9, r6, r7, lsl #4
55 and r11, r6, r6, lsl #4
56 eor r7, r10, r8
57 eor r6, r9, r11
58 // Iteration 4
59 and r8, r7, r4, lsl #8
60 and r9, r6, r4, lsl #8
61 eor r4, r9, r4
62 eor r4, r8, r4
63 and r10, r7, r5, lsl #8
64 and r11, r6, r5, lsl #8
65 eor r5, r10, r5
66 eor r5, r11, r5
67 orn r8, r7, r7, lsl #8
68 and r10, r7, r6, lsl #8
69 orn r9, r6, r7, lsl #8
70 and r11, r6, r6, lsl #8
71 eor r7, r10, r8
72 eor r6, r9, r11
73 // Post loop
74 and r9, r7, r4, lsl #16
75 and r8, r6, r4, lsl #16
76 eor r4, r9, r4
77 eor r4, r8, r4
78 and r11, r7, r5, lsl #16
79 and r10, r6, r5, lsl #16
80 eor r5, r11, r5
81 eor r5, r10, r5
82 eor r0, r2, r4, lsl #1
83 eor r1, r3, r5, lsl #1
```