Efficient Masking of ARX-Based Block Ciphers Using Carry-Save Addition on Boolean Shares*

Daniel Dinu, Johann Großschädl, and Yann Le Corre

SnT and University of Luxembourg
Laboratory of Algorithmics, Cryptology and Security (LACS)
6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
{dumitru-daniel.dinu,johann.groszschaedl,yann.lecorre}@uni.lu

Abstract. Masking is a widely-used technique to protect block ciphers and other symmetric cryptosystems against Differential Power Analysis (DPA) attacks. Applying masking to a cipher that involves both arithmetic and Boolean operations requires a conversion between arithmetic and Boolean masks. An alternative approach is to perform the required arithmetic operations (e.g. modular addition or subtraction) directly on Boolean shares. At FSE 2015, Coron *et al.* proposed a logarithmic-time algorithm for modular addition on Boolean shares based on the Kogge-Stone carry-lookahead adder. We revisit their addition algorithm in this paper and present a fast implementation for ARM processors. Then, we introduce a new technique for direct modular addition/subtraction on Boolean shares using a simple Carry-Save Adder (CSA) in an iterative fashion. We show that the average complexity of CSA-based addition on Boolean shares grows logarithmically with the operand size, similar to the Kogge-Stone carry-lookahead addition, but consists of only a single AND, an XOR, and a left-shift per iteration. A 32-bit CSA addition on Boolean shares has an average execution time of 162 clock cycles on an ARM Cortex-M3 processor, which is approximately 43% faster than the Kogge-Stone adder. The performance gain increases to over 55% when comparing the average subtraction times. We integrated both addition techniques into a masked implementation of the block cipher Speck and found that the CSA-based variant clearly outperforms its Kogge-Stone counterpart by a factor of 1.70 for encryption and 2.30 for decryption.

1 Introduction

The concrete security of a cryptographic system depends not only on the cryptanalytic complexity of the underlying algorithm, but also on the quality of its implementation. This became apparent some 20 years ago with the emergence of Side-Channel Analysis (SCA) [13], a special form of cryptanalysis that aims to exploit measurable physical phenomena (e.g. variations in the response time or power consumption) of a device executing a cryptographic algorithm so as to reveal information about the secret key. The most advanced variant of SCA is

* Supported by FNR Luxembourg (CORE project ACRYPT, ID C12-15-4009992).

Differential Power Analysis (DPA) [14], which involves two phases, namely an acquisition phase and an analysis phase. In the former phase, the attacker captures power consumption traces from the target device for different plaintexts or ciphertexts under the same secret key. Thereafter, in the analysis phase, she adopts sophisticated statistical techniques to determine the correlation between the power consumption and certain intermediate values that depend solely on the plaintext/ciphertext and (parts of) the secret key. Numerous case studies reported in the literature confirm that DPA attacks pose a real-world threat to the security of unprotected (or insufficiently protected) cryptosystems and can be mounted in relatively short time with relatively cheap equipment.

From a high-level point of view, countermeasures to thwart DPA attacks on symmetric cryptosystems can be broadly divided into two main categories; one is *hiding* (i.e. eliminating the data-dependency of the power consumption) and the second is *masking* (i.e. randomizing the intermediate values that are computed) [16]. Common approaches for hiding countermeasures aim to make the device's power consumption profile either constant for all possible values of the secret key or fully random (i.e. statistically independent from the key). Hiding can be implemented in hardware (e.g. by using a so-called DPA-resistant logic style) and software (e.g. by randomly shuffling the order in which the sensitive operations are executed or through the insertion of dummy operations) [16]. In both cases, the intention is to break (or, at least, to obscure) the link between the sensitive intermediate values that are computed during the execution of an algorithm and the power consumption traces. Masking, on the other hand, aims to conceal every key-dependent variable with a random value, called "mask," in order to break the link between the intermediate values that are computed on the device and the (unmasked) intermediate values of the algorithm. The basic principle is related to the idea of secret sharing since every sensitive variable is split into $n \geq 2$ "shares," so that any combination of up to $d = n - 1$ shares is statistically independent of any secret value. These n shares must be processed separately during the execution of the algorithm and then re-combined in the end to yield the correct result. When implemented properly, masking forces an attacker to combine n leakages originating from the n shares in order to obtain the secret information.

Depending on the actual operation to be protected against DPA, a masking scheme can be Boolean (using logical XOR), arithmetic (using modular addition or modular subtraction) or multiplicative (using modular multiplication). When a cryptographic algorithm involves arithmetic and Boolean operations, which is generally the case for all ARX-based block ciphers, then the masks have to be converted from one form to the other without introducing any kind of leakage [21]. Goubin was the first to describe secure algorithms for conversion between arithmetic and Boolean masks in [9]. While his method is very efficient for the Boolean-to-arithmetic conversion, it introduces a high overhead for conversions in the other direction. Coron and Tchulkine [5], as well as Debraize [6], came up with improved variants of Goubin's algorithm for switching from arithmetic to Boolean masking. At FSE 2015, Coron *et al.* [4] introduced a novel conversion

technique with logarithmic complexity based on a special "parallel-prefix" form of a carry-lookahead adder, known as *Kogge-Stone Adder (KSA)* [15]. Besides mask conversion, there exists a second principal approach for efficient masking of ARX-based ciphers, namely to perform the necessary arithmetic operations (e.g. modular addition/subtraction) directly on Boolean shares. This idea was originally proposed for hardware implementation [1], but can also be applied to protect software implementations of ARX-based block ciphers against DPA as demonstrated in [12]. The latency of the implementations in [1] and [12] grows linearly with the bit-length of the two operands. However, Coron *et al.* showed in [4] that the KSA allows not only logarithmic-time mask conversion, but also logarithmic-time modular addition on Boolean shares.

The KSA for modular addition on Boolean shares introduced in [4] comes with a formal security proof embedded into the framework of Isai, Sahai, and Wagner [11]. Furthermore, the authors of [4] present a software implementation of their addition technique written in ANSI C and analyzed its execution time on a 32-bit processor. They also report the execution time of first-order secure implementations of HMAC-SHA1 and the Speck cipher [2]. Unfortunately, an implementation written in C is not suitable for performance evaluations since optimizations introduced by the compiler may break the security of a masking scheme, even if the source code looks perfectly sound. On the other hand, when preventing a compiler from performing sophisticated optimizations, the results are not meaningful. Therefore, it is still unclear how fast a modular addition on Boolean shares can be in the real world and how its execution time impacts the performance of an ARX-based cipher. Another important question arising from [4] is whether there exists an alternative addition technique that could lead to better execution times than the KSA. Based on the work described in the present paper, we can answer this question positively. We propose a new technique for performing modular additions and subtractions directly on Boolean shares that uses a basic *Carry-Save Adder (CSA)* [17] in an iterative fashion, which is not only faster but also smaller (in terms of code size) than the KSA.

A masking scheme that uses the proposed CSA-based addition on Boolean shares is a lightweight countermeasure with relatively low impact on execution time and binary code size. The design of DPA countermeasures always involves a trade-off between security (i.e. the achieved "degree" of DPA resistance) and performance/resource requirements (RAM footprint, code size). Such trade-offs yield a wide spectrum of countermeasures along an axis between security and efficiency, whereby most existing proposals (including the KSA-based masking from [4]) are at the far end towards security. These countermeasures were typically developed for smart card applications where the secret key is fixed and an attacker can measure an arbitrary number of traces. Such applications require advanced DPA countermeasures, which usually introduce massive overheads in execution time [16]. However, applications outside the smart card domain can have different threat models, different assumptions about the number of traces the attacker can measure, and different security requirements. For example, in the Internet of Things (IoT), the secret key used to encrypt the communication

Table 1. The cost (in number of elementary operations) of different secure operations.

Secure Operation	Cost
SecNot	1
SecXor	2
SecShift	4
SecShiftFill	5
SecAnd	8
SecOr	11

between two devices is often provisioned dynamically (e.g. through ephemeral ECDH key exchange) and the amount of transmitted data is, in general, small (e.g. up to a few kB), which means that at most a few hundred data blocks are encrypted with one and the same key. In this case, an attacker can just capture a few hundred power or EM traces. The proposed masking using CSA addition on Boolean shares is a (relatively) inexpensive DPA countermeasure that can meet certain relaxed security requirements at significantly lower cost than the sophisticated countermeasures used for smart cards.

2 Preliminaries

A first step towards masked implementations of ARX-based ciphers is to define "secure" (i.e. masked) variants of the used arithmetic/logical operations: modular addition and subtraction, rotations, and bitwise exclusive OR. All bitwise logical operations and shifts (including rotations) are relatively easy to perform directly on Boolean shares, whereas the non-linear addition/subtraction require more complex algorithms. Coron *et al.* [4] presented a provably-secure method to perform a modular addition on Boolean shares using only secure algorithms for AND, XOR and bit shifts.

We specify in Table 1 all secure operations required to mask an ARX design and their cost expressed in the number of "elementary" operations, which can normally be executed via a single instruction. SecAnd, SecShift, and SecXor are described in detail in [4, Sect. 4]. Besides these, we need provably-secure algorithms for two further operations: SecOr and SecShiftFill. The former computes an OR on Boolean shares, while SecShiftFill shifts a sensitive value represented by Boolean shares n bit-positions to the left and fills the n least significant bits with 1 (see [7] for a more detailed treatment). We divide the secure operations on Boolean shares into three classes according to their computational cost. The first class includes all secure operations with a cost of at most six instructions (e.g. SecXor, SecShift). Then, the second class contains operations that can be masked using up to a dozen instructions (e.g. SecAnd, SecOr). Finally, the third class is represented by operations that need more than 12 instructions. Secure algorithms for modular addition/subtraction on Boolean shares belong to this latter class since they rely on secure operations from the first two classes.

Algorithm 1. Kogge-Stone Addition

Input: Operands $a, b \in \{0,1\}^k$
Output: Result $r = a + b \mod 2^k$
1: $p \leftarrow a \oplus b$
2: $g \leftarrow a \wedge b$
3: **for** i from 1 to $\max\big(\lceil \log_2(k-1) \rceil, 1\big)$ **do**
4: $\quad g \leftarrow \big(p \wedge (g \ll 2^{i-1})\big) \oplus g$
5: $\quad p \leftarrow p \wedge (p \ll 2^{i-1})$
6: **end for**
7: $g \leftarrow \big(p \wedge (g \ll 2^{n-1})\big) \oplus g$
8: $r \leftarrow a \oplus b \oplus (g \ll 1)$
9: **return** r

The Kogge-Stone Adder (KSA) [15] belongs to the family of parallel-prefix carry-lookahead adders, which parallelize the computation of the carry signal in order to reduce the carry propagation delay. The structure of a parallel-prefix adder can be represented through prefix graphs that generate at each stage two signals: a propagate signal p and a generate signal g. The KSA is very fast due to its minimal depth (which grows logarithmically with respect to the size of the operands) and minimal fan-out, but has a high node count and, thus, it suffers from wiring congestion when implemented in hardware.

The structure of the KSA can be easily parallelized in software as specified in Algorithm 1. If the adder does not get an input carry signal along with the two operands a and b, then the bitwise ORs can be replaced by bitwise XORs as in Algorithm 1. The addition on Boolean shares benefits tremendously from this optimization because the secure SecXor operation is much faster than the secure SecOr operation. Unfortunately, this optimization can not be applied to the subtraction (i.e. two's complement addition) because the input carry signal has to be set to 1 and distributed to all stages of the adder. Hence, a software implementation of KSA subtraction needs to fill the least significant bits of the generate word g with the value of the input carry after each left-shift. This leads to less efficient software implementations of subtraction versus addition.

3 Carry-Save Addition

The design of algorithms and respective hardware architectures for the addition of integers is one of the central research topics in computer arithmetic and has a history stretching back more than 50 years [18]. The efficiency of the various techniques proposed in the literature depends to a large extent on how the two operands to be added are represented. In the most basic case, i.e. the standard binary system, one uses a number representation radix of $r = 2$ and the digit set $D = \{0, 1\}$, which means a k-bit integer a is given as

$$a = \sum_{i=0}^{k-1} a_i \, 2^i \quad \text{with} \quad a_i \in \{0, 1\} \tag{1}$$

Throughout this paper, we shall use indexed lowercase letters to denote the individual bits of an integer (a_0 is the least significant bit of a and a_{k-1} is its most significant bit). The most basic way of adding up two k-bit integers is to apply a so-called *Ripple-Carry Adder (RCA)* consisting of k Full Adders (FAs) [18]. Each FA gets besides the two operand bits a_i and b_i also a carry bit c_{in} as input and produces a sum bit s_i and an outgoing carry bit c_{out} as follows.

$$s_i = a_i \oplus b_i \oplus c_{in}, \quad c_{out} = (a_i \wedge b_i) \vee (a_i \wedge c_{in}) \vee (b_i \wedge c_{in}) \tag{2}$$

The carry output c_{out} of each FA is connected to the carry input c_{in} of the next-higher FA. When analyzing the latency of an RCA, one needs to take into account the maximum possible length of a carry chain. As defined in [18], the length of a carry chain is the number of bit positions from where the carry is generated up to (and including) where it is finally absorbed or annihilated. The longest possible carry chain of a k-bit RCA covers all k FAs since, in the worst case, a carry generated at the least significant position ripples all the way up to the most significant position. As a consequence, the latency of an RCA grows linearly with the operand size. However, a single carry chain of length k occurs only for very few combinations of operands as we will discuss further below. In the case of random inputs, one can normally (i.e. on average) expect to have several, but much shorter, carry chains. It was already shown in 1946 that, on average, the carry chains in a k-bit addition are $\log_2(k)$ bits long [3].

Although RCAs are easy to implement in hardware, they are rarely used in high-speed arithmetic circuits. The maximum frequency with which an RCA is capable to process operands is determined by the worst-case signal propagation path, which, in turn, is determined by the maximum length of the carry chains (i.e. k) and not their average length (i.e. $\log_2(k)$) [18]. This has motivated the development of advanced adder circuits having a worst-case latency that grows logarithmically with the operand length. A good example for such an advanced adder is the KSA described in the preceding section. A logarithmic worst-case behavior is the optimum that one can achieve with the binary number system [18]. However, when using a redundant number system (i.e. a number system with a digit set D containing more than r elements), it is even possible to add two integers in constant time, independent of their length.

A very important redundant number system is the *Carry-Save (CS)* system [17], which uses a radix-2 representation with the digit set $D = \{0, 1, 2\}$. Since any digit a_i can take three possible values (namely 0, 1, and 2), it needs to be encoded using two bits, a sum bit a_i^s and a carry bit a_i^c, as shown below.

$$0 \leftrightarrow (0,0) \qquad 1 \leftrightarrow (0,1) \text{ or } (1,0) \qquad 2 \leftrightarrow (1,1) \tag{3}$$

The actual value of a k-digit number a given in CS form is

$$a = \sum_{i=0}^{k-1} a_i \cdot 2^i = \sum_{i=0}^{k-1} (a_i^s + a_i^c) \cdot 2^i \text{ with } a_i^s, a_i^c \in \{0,1\} \tag{4}$$

A k-digit CS integer a is always composed of a sum-word a^s and a carry-word a^c, each of which consists of k bits. Thus, we can write $a^s = (a_{k-1}^s, \ldots, a_1^s, a_0^s)$

and $a^c = (a^c_{k-1}, \ldots, a^c_1, a^c_0)$. The redundancy in the digit set D, which enables two encodings for the digit 1, means that the CS representation of an integer is not unique [18]. An integer a given in CS representation can be converted into conventional binary form by simply adding up its sum-word a^s and carry-word a^c using e.g. an RCA or KSA, i.e. the redundant-to-binary conversion involves always a propagation of carries. In some way, the sum-word a^s and carry-word a^c can be interpreted as two *arithmetic shares* of the integer a since their sum $a^s + a^c$ is exactly a. In practice, the CS representation is typically used for the implementation of complex arithmetic operations that require a multi-operand addition; a typical example is the addition of partial products performed in an integer multiplication [18]. The CS representation is attractive for this purpose because it allows partial products to be added up in constant time, irrespective of k, yielding a result in CS form. Only at the end of a multiplication, a single carry-propagating addition is needed for the redundant-to-binary conversion.

Let a be a k-digit integer in CS form and b a binary integer of k bits. The result $r = a + b$ of a CS addition can be computed in parallel for all digits and consists of a sum-word r^s and a carry-word r^c, obtained as follows.

$$r^s_i = a^s_i \oplus a^c_i \oplus b_i \quad \text{for } 0 \le i \le k-1 \tag{5}$$

$$r^c_i = (a^s_{i-1} \wedge a^c_{i-1}) \vee (a^s_{i-1} \wedge b_{i-1}) \vee (a^c_{i-1} \wedge b_{i-1}) \quad \text{for } 1 \le i \le k \tag{6}$$

A *Carry-Save Adder (CSA)* can be easily implemented in hardware through an array of k FAs, similar to the RCA [18]. However, the carry-propagation in the CSA is limited to a single position, which becomes immediately evident from Eq. (6) because r^c_i depends solely on bits with index $i-1$. A carry generated by an FA just goes to the next-higher FA, but can not ripple up further. The overall latency of a k-digit CSA is, therefore, determined by the latency of an FA and does not depend on k anymore. The least significant bit of the result's carry-word, i.e. r^c_0, must be set to 0 when performing an addition, and to 1 in the case of a subtraction, as we will explain further below.

3.1 Using a CSA for Single-Operand Addition

Traditionally, CSAs are employed for multi-operand addition, i.e. in situations where many operands (e.g. partial products of an integer multiplication) are to be summed up. However, in the present paper we use a CSA to perform single-operand additions to add two k-bit integers, a and b, in standard binary form with the goal of obtaining a binary result. Computing the sum $r = a + b$ in CS form is easy and requires just a logical AND and a logical XOR operation:

$$r^s_i = a_i \oplus b_i \quad \text{for } 0 \le i \le k-1 \tag{7}$$

$$r^c_i = a_{i-1} \wedge b_{i-1} \quad \text{for } 1 \le i \le k \tag{8}$$

An arithmetic circuit computing r^s_i and r^c_i according to the equations above is commonly referred to as a Half-Adder (HA). Similar as before, i.e. Eqs. (5) and (6), the sum bits r^s_i are kept "in place," whereas all the carry bits r^c_i move one

Algorithm 2. Carry-Save Addition

Input: Operands $a, b \in \{0,1\}^k$
Output: Result $r = a + b \bmod 2^k$
1: $t \leftarrow a \wedge b$
2: $r^s \leftarrow a \oplus b$
3: $r^c \leftarrow t \ll 1$
4: **while** $r^c \neq 0$ **do**
5: $\quad t \leftarrow r^s \wedge r^c$
6: $\quad r^s \leftarrow r^s \oplus r^c$
7: $\quad r^c \leftarrow t \ll 1$
8: **end while**
9: **return** r^s

position to the left. Since, in this paper, additions and subtractions are always done modulo 2^k, we can simply discard the most significant carry bit r^c_k. When implemented in software, a HA consists of an AND instruction, an XOR, and a 1-bit left shift, which is a lot more efficient than the sequence of instructions carried out by the KSA (Algorithm 1). Another advantage of the CSA over the KSA is that a subtraction is only slightly more complex than an addition. The most common way to perform a subtraction $r = a - b$ is to add the two's complement of b to a. To generate the two's complement of b, we have to first form the one's complement (through an inversion of all bits of b) and then add 1 to it [18]. Fortunately, this addition of 1 can be simply realized by just setting the least significant carry bit r^c_0 to 1. Adding 1 in this way is always possible when using a CSA[1], but not with a KSA. Hence, a CS subtraction is essentially the same as a CS addition with inverted addend bits.

While the benefit of the CSA for multi-operand addition (multiplication) is clear, it may seem counterintuitive to use a CSA for a single addition since the result is obtained in CS form and still needs to be converted into the standard binary representation, which requires a propagation of carries. This raises the question of why one does not simply use a KSA or some similar kind of carry-propagating adder in the first place. The answer lies in the rather little-known fact that a CSA can not only be employed to perform a CS addition, but also for the redundant-to-binary conversion of the result. Namely, when we feed the sum-word r^s and carry-word r^c obtained through Eqs. (7) and (8) as input into a CSA, we get again a result in CS representation, but with fewer 1 bits in the carry-word r^c (i.e. lower Hamming weight) or no 1 bits at all. When repeating this procedure, all bits of the carry-word r^c will eventually become 0, and the latest sum-word r^s represents the result in binary form. In each iteration, the Hamming weight of r^c is reduced by (at least) 1 since the lowest carry bit r^c_0 is set to 0. Algorithm 2 specifies this addition technique in a formal fashion. The first three lines do the actual CS addition of the operands a and b according to Eqs. (7) and (8), yielding a sum in CS form consisting of r^s and r^c. Then, the sum is converted into standard binary representation using the while-loop.

[1] As mentioned before, r^c_0 is normally set to 0 when performing an addition.

The overall execution time of Algorithm 2 depends on the number of loop-iterations needed for the redundant-to-binary conversion of the result r, i.e. the number of iterations that have to be executed until the carry-word r^c becomes 0. Intuitively, one expects the number of iterations to be closely related to the average length of the carry chains that occur when a is added to b, which, as explained earlier in this section, is approximately $\log_2(k)$ for k-bit operands. In [10], Hendrickson experimentally assessed the accuracy of the $\log_2(k)$ approximation and concluded that $\log_2(5k/4)$ makes a better estimate for the average length of the carry chains. This suggests an average of around 4.3 bit positions for the carry-chain length when 16-bit operands are added, and about 5.3 bits in the case of $k = 32$. However, these results are not immediately applicable to the estimation of the number of loop iterations of Algorithm 2 since the carries are generated outside the loop (namely in the actual CS addition of a and b in line 1–3). Therefore, the number of iterations is one less than the length of the carry chains, i.e. based on Hendrickson's formula we can estimate the average number of iterations to be about $\log_2(5k/4) - 1$. In this way, we finally obtain $\log_2(20) - 1 \approx 3.3$ iterations if $k = 16$ and $\log_2(40) - 1 \approx 4.3$ iterations for the redundant-to-binary conversion when $k = 32$. Thus, the average execution time of the CSA addition specified in Algorithm 2 increases *logarithmically* with the operand length k, similar to the execution time of the KSA.

3.2 Security Aspects

Even though both the CSA and KSA have logarithmic time complexity, there exists a significant difference, namely that the execution time of the former is not constant for a given operand length. Based on above analysis, the average number of iterations of the while-loop for redundant-to-binary conversion can be approximated as $\log_2(5k/4) - 1$. In the best case, however, the while-loop is not iterated at all, which happens when $(a \wedge b) \ll 1$ is 0. On the other hand, in the worst case, a total of $k - 1$ iterations need to be performed until all k bits of the carry-word r^c become 0. The $k - 1$ iterations are the absolute maximum since, in each iteration, the least significant carry bit r^c_0 is set to 0. When the operands are short, e.g. when $k = 16$, it is feasible to (exhaustively) determine the *exact* number of iterations for all 2^{2k} combinations of input words. Figure 1 shows the probabilities of all possible iteration counts for $k = 16$, which ranges from 0 to 15. Out of the total of 2^{32} possible operand combinations, only some 1.34% (or 57,395,628 combinations to be precise) directly yield a final result in binary form (i.e. $r^c = 0$) such that the loop is not iterated at all. An iteration count of three has the highest probability; it occurs for roughly 27.72% of the input combinations, closely followed by two iterations with a probability in the area of 27.01%. The maximum possible 15 iterations happen only with 65,536 input combinations, i.e. the probability of the worst case for $k = 16$ amounts to only 2^{-16}, or roughly 0.0015%. The average over all 2^{32} possible combinations of pairs of 16-bit input words is approximately 3.25 iterations, which confirms that the estimated iteration-count of $\log_2(5k/4) - 1 \approx 3.3$ was pretty accurate and the same also holds for $k = 8$ as we experimentally verified.

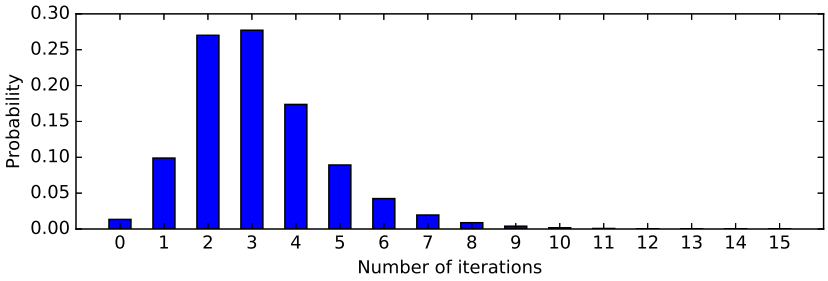

Fig. 1. Probability of each possible number of iterations of the loop for redundant-to-binary conversion when $k = 16$.

An attacker may be able to count the number of iterations executed in the redundant-to-binary conversion, which raises the question of what information the iterations reveal about the operands or the result. Let us first consider the scenario that the loop is not iterated at all, which can only happen if after the left-shift operation (line 3 of Algorithm 2), the carry-word r^c is 0. This means an attacker learns that $a \wedge b$ is either 0 or 2^{k-1} because only in these two cases r^c can become 0. Answering the question of what information the attacker can learn about the obtained result r (i.e. the sum-word r^s) in this scenario is less obvious and boils down to the question of whether r can take all 2^k possible values between 0 and $2^k - 1$ or not. Due to our experiments with $k = 16$ from above, we know already that there are 57,395,628 different input combinations for which the loop is not executed at all. A further analysis reveals that these combinations cover all 2^{16} possible values for the resulting sum, i.e. r can have any value between 0 and $2^{16} - 1$. However, this does not hold any longer when the redundant-to-binary conversion consists of exactly one loop iteration since now only $2^{16} - 2 = 65{,}534$ different values for the sum can be obtained. In the most extreme case, i.e. when the maximum number of $k - 1 = 15$ iterations is performed, the resulting sum can only be either 0 or 2^{15}, which means that an attacker has a 50% chance to simply guess the value of the sum. However, the probability of 15 iterations is extremely small, namely $2^{-16} \approx 0.0015\%$.

The above analysis of the iteration counts is based on an exhaustive testing of all possible pairs of input words, which is feasible for $k = 16$, but not when $k = 32$ anymore since the number of combinations of 32-bit words amounts to 2^{64}. However, one can expect that for $k = 32$, the distribution of probabilities for iteration counts will be similar to Fig. 1, meaning the highest probabilities are centered around four iterations and the probability of $k - 1 = 31$ iterations is extremely small, namely 2^{-32}. Concretely, an attacker would have to observe on average some $4.29 \cdot 10^9$ CS additions to reduce the guessing entropy for the result to 1. However, the attacker can "trade" the number of guesses she has to make for the number of traces she has to acquire to mount a DPA attack. The relation between the guessing entropy and the number of iterations, as well as the probability of each of the 32 iteration counts, is graphically represented in

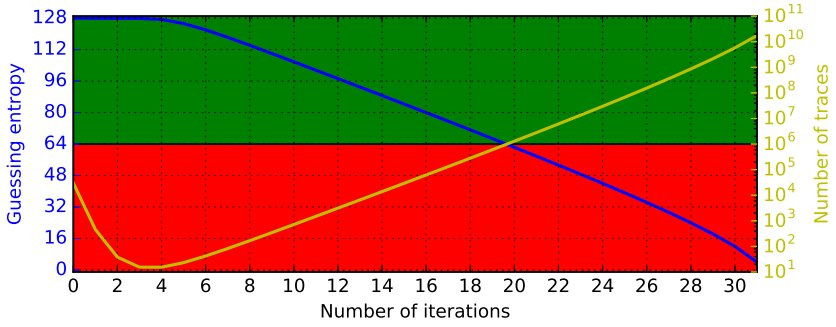

Fig. 2. The number of traces necessary to reduce the guessing entropy of a 128-bit key for different number of iterations of four 32-bit CS additions protected against DPA.

Fig. 2. We generated the information shown in this figure through experiments with 2^{46} pairs of 32-bit words since exhaustively testing all 2^{64} combinations is not feasible. Concretely, in these experiments, we added a 128-bit secret key to a 128-bit state consisting of four 32-bit words.

Due to our experiments, we can confirm that an iteration count of four has the highest probability among the 32 possible counts and occurs in about 25% of the 32-bit CS additions. In a real DPA attack on a 128-bit key addition, one can therefore expect approximately one out of 16 power consumption traces to contain four additions with four iterations each. As indicated by the yellow line in Fig. 2, an exponentially increasing number of power traces must be captured as the number of iterations gets larger. For example, an attacker would need to measure (at least) 10^6 power traces before she can expect to encounter a trace with more than 20 iterations. The blue line in Fig. 2 shows that the larger the number of iterations gets, the more information about the sum is "leaked." As analyzed before for $k = 16$, the number of distinct values that the result of an addition can take decreases with the number of loop iterations executed in the redundant-to-binary conversion. In the most extreme case of $k - 1$ iterations (which happens with a very small probability of 2^{-32} for $k = 32$), the result can only take two distinct values, namely 0 and 2^{k-1}. Fortunately, the restriction of the value space for the result is much lower for the iteration counts with the highest probabilities, which are centered around four iterations. Nonetheless, it is possible to exploit the number of iterations in our experiments to reduce the guessing entropy for the 128-bit key. For example, our results show that if an attacker is able to observe some 600,000 additions, the guessing entropy would be reduced from 128 to 64 bits. Similarly, the ability to measure power traces of roughly 62,000 additions would reduce the guessing entropy to 80 bits. However, the assumption that an attacker is capable to measure power traces from several 10,000 or even 100,000 encryptions with one and the same key may be reasonable for smart cards, but is extremely unrealistic in many other contexts (we discussed in Sect. 1 secure communication in the IoT as an example where secret keys are ephemeral and only used to encrypt small amounts of data).

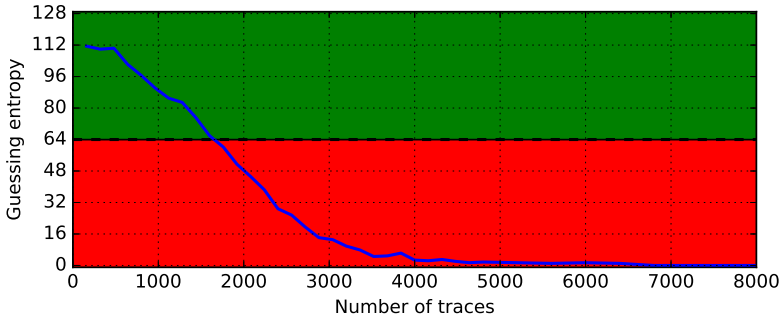

Fig. 3. The evolution of the guessing entropy for a Correlation Power Analysis (CPA) attack against four unprotected 32-bit modular additions.

The best attack against a protected (i.e. masked) version of the CSA takes the number of loop-iterations performed in the redundant-to-binary conversion into account to reduce the guessing entropy of the key. This raises the question of how much harder a protected CSA implementation is to attack in relation to an unprotected 32-bit modular addition. To answer this question, we mounted a Correlation Power Analysis (CPA) attack against an unprotected implementation of modular addition executed on an ARM Cortex-M3 processor clocked at 33 MHz. The result, depicted in Fig. 3, shows that four 32-bit additions can be attacked with some 6,800 power traces on average. On the other hand, when using a similar amount of traces (namely 7,467), to attack a protected implementation of the CSA, the guessing entropy of a single 32-bit addition can be reduced by only 5.18 bits. Therefore, the protected CSA considerably increases the attacker's effort compared to an unprotected modular addition.

The leakage caused by the operand-dependent number of loop-iterations in the redundant-to-binary conversion reduces the guessing entropy of the secret key and, hence, the effective security level, depending on the number of traces an attacker is able to capture. However, for an effective security level of e.g. 96 bits, a masked implementation of 128-bit Speck (i.e. Speck-64/128) based on the protected CSA is still much faster (namely about 17.5% for encryption and roughly 42.4% for decryption) than a masked implementation of 96-bit Speck (i.e. Speck-64/96) using the protected KSA, as we will see in Subsect. 4.2.

Although the protected CSA could be applied to operands of any size, the trade-off between the number of traces and the guessing entropy must be taken into account. This trade-off must be particularly carefully analyzed for operand lengths below 32 bits. As a general guideline, we only recommend the protected CSA for operands with a bitlength of $k \geq 32$ as otherwise the security margin might become too tight. Yet, even with this restriction, the protected CSA can be used to efficiently mask numerous cryptographic primitives that are used in practice, including a diverse range of ARX-based lightweight symmetric ciphers (like Chaskey, Speck, RECTANGLE, LEA, etc.) and the keyed-hash message authentication code based on SHA-256 (HMAC-SHA-256).

Table 2. Execution time and code size of secure addition on Boolean shares using the secure Kogge-Stone Adder (KSA) and the secure Carry-Save Adder (CSA). Since the execution time of the CSA is not constant, we specify the average number of cycles over 100,000 executions with random inputs.

Adder	Time (cycles)		Code size (bytes)	
	Addition	Subtraction	Addition	Subtraction
KSA rolled	282	369	292	408
KSA unrolled	202	291	544	808
CSA (average)	161.75	165	136	148

4 Implementation Details and Evaluation

We implemented secure addition/subtraction on Boolean shares using both the KSA and CSA algorithm in Assembly language for a 32-bit ARM Cortex-M3 processor. Then, we applied the mentioned addition techniques to protect two variants of the block cipher Speck [2] against first-order DPA attacks.

4.1 Secure Addition on Boolean Shares

The implementation results for secure addition/subtraction on Boolean shares are shown in Table 2. Before discussing the results in detail, we briefly describe the implementations. Like other 32-bit ARM processors, the Cortex-M3 has 13 general-purpose registers, which we allocate as follows: Four registers hold the shares of the two masked inputs. Either two or three registers (depending on the algorithm) store the randomly generated 32-bit values needed for the execution of the secure Boolean functions like SecureAnd and SecureShift. Each algorithm also occupies a certain number of registers for intermediate results: three in the case of the CSA and four for the KSA. A special property of ARM processors is their ability to execute a shift operation together with most arithmetic/logical instructions within a single clock cycle. We exploited this feature to reduce the execution time of both the CSA-based and KSA-based addition technique.

The secure KSA performs additions in constant time and can be implemented with either "rolled" or unrolled loops. The entirely unrolled version of the KSA is between 28% (addition) and 21% (subtraction) faster than a standard implementation with rolled loops, but this gain in speed comes at the expense of almost doubling the binary code size. In both cases, the KSA subtraction is significantly slower than the addition because the SecureXor operation has to be replaced by the less efficient SecurOr and the left-shifts by n bits performed on the shares of the generate word require the insertion of n bits set to 1 (i.e. the SecShift must be replaced by SecShiftFill). However, in the unrolled version, the execution time of SecShiftFill can be sightly reduced by using immediate values instead of registes (see [7] for further implementation details).

The secure CSA is very efficient thanks to its simple structure that involves only SecXor, SecAnd, and SecShift operations. Unlike the "rolled" version of the KSA, it does not need a separate register to hold a loop counter. However, the

Table 3. Execution time, code size and performance penalty factor of different implementations of Speck-64/96 and Speck-64/128. Since the execution time of the CSA is not constant, we specify the average number of cycles over 100,000 executions with random inputs.

Implementation	Time (cycles)		Code size (B)		Penalty factor	
	Enc	Dec	Enc	Dec	Enc	Dec
Unprotected Speck-64/96	306	510	44	52	1	1
Speck-64/96 (KSA rolled)	6639	9525	340	480	21.69	18.67
Speck-64/96 (KSA unrolled)	4921	7447	592	876	16.08	14.60
Speck-64/96 (CSA average)	3902.9	4071.8	180	204	12.75	7.98
Unprotected Speck-64/128	318	530	44	52	1	1
Speck-64/128 (KSA rolled)	6892	9889	340	480	21.67	18.65
Speck-64/128 (KSA unrolled)	5108	7731	592	876	16.06	14.58
Speck-64/128 (CSA, average)	4061.3	4290.8	180	204	12.77	8.09

main advantage of the CSA over the KSA is that a subtraction is only slightly slower than an addition since it requires just two extra operations, namely an inversion and the insertion of a 1 at the LSB-position of the carry-word.

A direct comparison of the results of the rolled version of the KSA and the CSA allows us to conclude that the carry-save approach is not only faster, but also notably smaller than the Kogge-Stone technique. While the CSA addition is, on average, about 43% faster than the KSA addition (162 vs. 282 cycles as per Table 2), the difference increases to some 55% for subtraction (165 vs. 369 cycles). The benefit of the CSA over the KSA is even more significant in terms of code size since the difference amounts to a factor of about 2.14 for addition and 2.75 for subtraction. However, as we mentioned before, the execution time of the KSA can be improved by full loop unrolling, but the resultant code-size penalty may be undesirable for certain highly constrained environments where every single byte matters. In summary, using the proposed carry-save technique to directly perform a modular addition or subtraction on Boolean shares shows clear speed and size advantages over the KSA.

4.2 Masked Implementation of Speck

Speck is a family of lightweight block ciphers designed by cryptographers from the U.S. National Security Agency [2]. Speck-64/128 uses a two-branch Feistel network to encrypt 64-bit plaintexts with a 128-bit master key. Its round function is iterated 26 times and consists of simple operations on 32-bit words: two rotations, a modular addition, and two XORs. In the case of a straightforward (i.e. unprotected) implementation, the cipher's state fits into two registers, and a third register is needed for the round key. The remaining eleven registers are available for other purposes, e.g. the implementation of a masking technique to protect the cipher against DPA attacks.

The implementation results presented in Table 3 show that the unprotected version of Speck-64/128 is quite efficient compared to the secure addition on

Boolean shares (Table 2). Concretely, the encryption time is just a little worse than the execution time of the slowest addition on Boolean shares (i.e. rolled KSA), while the code size is at least six times smaller than the code size of the KSA. The code size of the unprotected implementation of Speck is also more than three times smaller than the size of the CSA, which is the adder with the smallest footprint. This high-level comparison clearly illustrates the enormous cost of masking just a single nonlinear operation, the modular addition. It can therefore be expected that the integration of masking will entail a massive performance degradation and also inflate the code size. Hence, any effort spent on optimizing masking is well spent. Even a modest improvement by a few cycles when performing a masked addition has the potential to yield a non-negligible overall performance gain.

A masked implementation of Speck occupies four registers to store the two shares of the 64-bit state. Depending on the implementation methodology, one or two registers have to be used to manipulate the shares of the round key. An ARM Cortex-M3 processor does not provide enough general-purpose registers to hold all operands needed during the execution of a masked implementation of Speck-64/128 using the secure KSA algorithm. Thus, at the beginning of an addition (or subtraction), two registers have to be spilled to RAM so that the necessary number of registers becomes available for the KSA. The original content of these registers is recovered at the end of the operation. However, for the fully unrolled implementation, it suffices to save only a single register onto the stack. These stack instructions (i.e. `push` and `pop`) add quite some overhead to each execution of the secure KSA-based modular addition/subtraction. On the other hand, the protected implementations of Speck-64/128 using the secure CSA are able to execute all operations directly on registers (i.e. no `push`/`pop` is required) since the underlying algorithm operates on fewer variables.

We compare in Table 3 the execution time and code size of an unprotected implementation of Speck-64/96 and Speck-64/128 with three DPA-protected versions[2]. All implementations have received a similar amount of optimization and perform a single iteration of the round function in a loop, i.e. we refrained from full loop unrolling to keep the code size small. The penalty factor on the execution time of Speck-64/128 introduced by the different masking schemes varies between 8.09 and 21.67. As expected, the efficiency of the three variants reflects the performance of the underlying method for addition or subtraction on Boolean shares. When comparing the masked implementation based on the CSA with the two KSA versions, it turns out that the encryption time of the former is more than 20% better than that of the unrolled KSA and 3.28 times smaller in size. The performance gain even doubles to 41% (which corresponds to a considerable speed-up factor of 1.70) when we compare the CSA with the rolled-loop KSA version. Furthermore, the CSA-based Speck implementation clearly outperforms its two KSA-based Speck counterparts in decryption; it is 1.80 times faster than the unrolled KSA variant (4291 vs. 7731 cycles) and 2.30 times faster than the KSA with a "rolled" loop (9889 cycles).

[2] The results exclude the generation of (pseudo-)random numbers for masking.

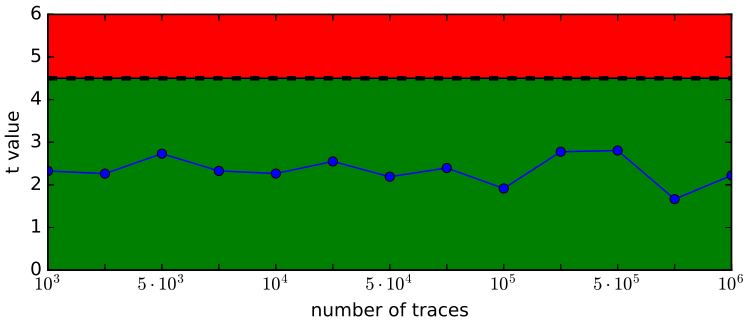

Fig. 4. T-statistic (absolute values) of the CSA under a HW leakage model.

4.3 Leakage Assessment

We evaluated the DPA-protected implementation of CSA addition on Boolean shares as well as the masked implementation of SPECK based on the CSA using Welch's t-test [8] on simulated power traces. Doing the test on simulated traces facilitates experiments with a large number (e.g. millions) of traces and reduces time and memory complexity in relation to real measurements. Our evaluation framework is inspired by the tool described in [19], but applies a "fixed versus random" leakage detection methodology. We eliminated the leakage related to the number of loop iterations performed in the redundant-to-binary conversion by simply executing the maximum number of iterations (i.e. 31) for all possible combinations of 32-bit input words. Then, we judiciously applied the t-test to avoid any wrong outcome [20]. Yet, we did not observe any significant leakage above the ± 4.5 threshold (which corresponds to a high statistical significance level of $\alpha = 0.001$) in our evaluation.

The maximum absolute value of the t-statistic of the secure implementation of the CSA addition is graphically represented in Fig. 4 for different numbers of simulated traces under a Hamming Weight (HW) leakage model. The t value is always well below the threshold of 4.5 in all our experiments and only shows small variations when increasing the number of traces from 10^3 to 10^6. To give a concrete example, the result of the t-test applied to 10^6 power traces of the secure CSA addition is depicted in Fig. 5. Again, we can observe that the value of the t-statistic is inside the ± 4.5 interval for each point in time, which implies that the null hypothesis holds. In other words, the masking scheme is effective against first-order DPA attacks because it passes the t-test evaluation. All these results strongly indicate that the implementation will also not leak when more than 10^6 traces are used for the t-test. Therefore, the described implementation of CSA addition can be deemed secure against first-order DPA attacks.

We obtained similar results when we applied the t-test to the secure implementation of KSA addition on Boolean shares and the masked implementation of the SPECK cipher based on the KSA. This suggests that our implementation of the KSA can be considered secure against first-order DPA attacks.

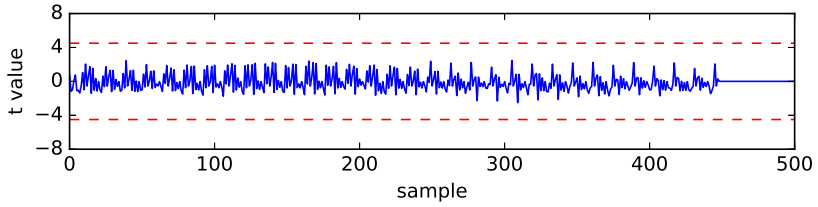

Fig. 5. The result of the t-test applied to the CSA under a HW leakage model.

5 Conclusions

The implementation of lightweight symmetric cryptosystems requires a careful balance between efficiency and security, including a certain degree of resistance against DPA attacks. In this context, we introduced a new masking technique for block ciphers that involve both arithmetic and Boolean operations, which is the case for Speck and many other ARX designs. Our main contribution is an algorithm for performing CSA-based modular addition/subtraction directly on Boolean shares, which makes expensive mask conversions obsolete. The CSA is much simpler and, hence, faster than the KSA presented at FSE 2015, but has operand-dependent execution time. Concretely, a CSA-based 32-bit addition on Boolean shares requires 162 clock cycles when executed on an ARM Cortex-M3 processor, which is between 20% and 41% faster than the KSA, depending on whether the loops are unrolled or not. We integrated both addition techniques into a masked implementation of Speck and found the CSA to outperform the "looped" KSA by a factor of 1.70 for encryption and 2.30 for decryption. The main drawback of the CSA is its operand-dependent execution time, which can be exploited to reduce the guessing entropy of the secret key. Nonetheless, the CSA is a practical and useful alternative to the KSA, especially for applications that encrypt only small amounts of data with one and the same key.

References

1. Y.-J. Baek and M.-J. Noh. Differential power attack and masking method. *Trends in Mathematics*, 8(1):53–67, June 2005.
2. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, 2013. Available for download at `http://eprint.iacr.org/2013/404`.
3. A. W. Burks, H. H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Report to U.S. Army Ordnance Department, 1946. Available for download at `http://library.ias.edu/files/Prelim_Disc_Logical_Design.pdf`.
4. J.-S. Coron, J. Großschädl, M. Tibouchi, and P. K. Vadnala. Conversion from arithmetic to Boolean masking with logarithmic complexity. In *Fast Software Encryption — FSE 2015*, vol. 9054 of *Lecture Notes in Computer Science*, pp. 130–149. Springer Verlag, 2015.

5. J.-S. Coron and A. Tchulkine. A new algorithm for switching from arithmetic to Boolean masking. In *Cryptographic Hardware and Embedded Systems — CHES 2003*, vol. 2779 of *Lecture Notes in Computer Science*, pp. 89–97. Springer Verlag, 2003.
6. B. Debraize. Efficient and provably secure methods for switching from arithmetic to Boolean masking. In *Cryptographic Hardware and Embedded Systems — CHES 2012*, vol. 7428 of *Lecture Notes in Computer Science*, pp. 107–121. Springer Verlag, 2012.
7. D. Dinu. *Efficient and Secure Implementations of Lightweight Symmetric Cryptographic Primitives*. PhD thesis, University of Luxembourg, 2017.
8. G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi. A testing methodology for side-channel resistance validation. In *Proceedings of the NIST Non-Invasive Attack Testing Workshop (NIAT 2011)*, pp. 158–172, Sept. 2011. Available for download at `http://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf`.
9. L. Goubin. A sound method for switching between Boolean and arithmetic masking. In *Cryptographic Hardware and Embedded Systems — CHES 2001*, vol. 2162 of *Lecture Notes in Computer Science*, pp. 3–15. Springer Verlag, 2001.
10. H. C. Hendrickson. Fast high-accuracy binary parallel addition. *IRE Transactions on Electronic Computers*, 9(4):465–469, Dec. 1960.
11. Y. Ishai, A. Sahai, and D. A. Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology — CRYPTO 2003*, vol. 2729 of *Lecture Notes in Computer Science*, pp. 463–481. Springer Verlag, 2003.
12. M. Karroumi, B. Richard, and M. Joye. Addition with blinded operands. In *Constructive Side-Channel Analysis and Secure Design — COSADE 2014*, vol. 8622 of *Lecture Notes in Computer Science*, pp. 41–55. Springer Verlag, 2014.
13. P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology — CRYPTO '96*, vol. 1109 of *Lecture Notes in Computer Science*, pp. 104–113. Springer Verlag, 1996.
14. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology — CRYPTO '99*, vol. 1666 of *Lecture Notes in Computer Science*, pp. 388–397. Springer Verlag, 1999.
15. P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, 22(8):786–793, Aug. 1973.
16. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Verlag, 2007.
17. G. Metze and J. E. Robertson. Elimination of carry propagation in digital computers. In *Proceedings of the International Conference on Information Processing, Paris, France, June 15-20, 1959*, pp. 389–395. UNESCO, 1960.
18. B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000.
19. O. Reparaz. Detecting flawed masking schemes with leakage detection tests. In *Fast Software Encryption — FSE 2016*, vol. 9783 of *Lecture Notes in Computer Science*, pp. 204–222. Springer Verlag, 2016.
20. F.-X. Standaert. How (not) to use Welch's t-test in side-channel security evaluations. Cryptology ePrint Archive, Report 2017/138, 2017. Available for download at `http://eprint.iacr.org/2017/138`.
21. P. K. Vadnala and J. Großschädl. Faster mask conversion with lookup tables. In *Constructive Side-Channel Analysis and Secure Design — COSADE 2015*, vol. 9064 of *Lecture Notes in Computer Science*, pp. 207–221. Springer Verlag, 2015.