Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity[*]

Jean-Sébastien Coron[1], Johann Großschädl[1], Mehdi Tibouchi[2], and Praveen Kumar Vadnala[1]

[1] University of Luxembourg,
{jean-sebastien.coron,johann.groszschaedl,praveen.vadnala}@uni.lu
[2] NTT Secure Platform Laboratories, Japan
tibouchi.mehdi@lab.ntt.co.jp

Abstract. A general technique to protect a cryptographic algorithm against side-channel attacks consists in masking all intermediate variables with a random value. For cryptographic algorithms combining Boolean operations with arithmetic operations, one must then perform conversions between Boolean masking and arithmetic masking. At CHES 2001, Goubin described a very elegant algorithm for converting from Boolean masking to arithmetic masking, with only a *constant* number of operations. Goubin also described an algorithm for converting from arithmetic to Boolean masking, but with $\mathcal{O}(k)$ operations where k is the addition bit size. In this paper we describe an improved algorithm with time complexity $\mathcal{O}(\log k)$ only. Our new algorithm is based on the Kogge-Stone carry look-ahead adder, which computes the carry signal in $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$ for the classical ripple carry adder. We also describe an algorithm for performing arithmetic addition modulo 2^k directly on Boolean shares, with the same complexity $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$. We prove the security of our new algorithm against first-order attacks. Our algorithm performs well in practice, as for $k = 64$ we obtain a 23% improvement compared to Goubin's algorithm. Our solution naturally extends to higher-order countermeasures with complexity $\mathcal{O}(n^2 \cdot \log k)$ instead of $\mathcal{O}(n^2 \cdot k)$ for n shares.

1 Introduction

Side-channel attacks. Side-channel attacks belong to the genre of implementation attacks and exploit the fact that any device performing a cryptographic algorithm leaks information related to the secret key through certain physical phenomena such as execution time, power consumption, EM radiation, etc. Depending on the source of the information leakage and the required post-processing, one can distinguish different categories of side-channel attacks, e.g. timing attacks, Simple Power Analysis (SPA) attacks, and Differential Power Analysis (DPA) attacks [KJJ99]. The former uses data-dependent (i.e. plaintext-dependent) variations in the execution time of a cryptographic algorithm to deduce information about the secret key involved in the computation of the ciphertext. In contrast, power analysis attacks require the attacker to measure the power consumption of a device while it executes a cryptographic algorithm [PMO07]. To perform an SPA attack, the attacker typically collects only one (or very few) power trace(s) and attempts to recover the secret key by focusing on differences between patterns within a trace. A DPA attack, on the other hand, requires many power traces and employs sophisticated statistical techniques to analyze differences between the traces [MOP07].

Even though DPA was first described using the DES algorithm as an example, it became soon clear that power analysis attacks can also be applied to break other secret-key algorithms, e.g. AES as well as public-key algorithms, e.g. RSA. A DPA attack normally exploits the principle of divide and conquer, which is possible since most block ciphers use the secret key only partially at a given point of time. Hence, the attacker can recover one part of the key at a time by studying the relationship between the actual power consumption and estimated power values derived from a theoretical model of the device. During the past 15 years, dozens of papers about successful DPA attacks on different implementations (hardware, software) of numerous secret-key cryptosystems (block ciphers, stream ciphers, keyed-hash message authentication codes) have been published. The experiments described in these papers confirm the real-world impact of DPA attacks in the sense

[*] An extended abstract will appear at FSE 2015; this is the full version.

that unprotected (or insufficiently protected) implementations of cryptographic algorithms can be broken in relatively short time using relatively cheap equipment.

The vast number of successful DPA attacks reported in the literature has initiated a large body of research on countermeasures. From a high-level point of view, countermeasures against DPA attacks can be divided into *hiding* (i.e. decreasing the signal-to-noise ratio) and *masking* (i.e. randomizing all the sensitive data) [MOP07]. Approaches to hiding-style countermeasures attempt to "equalize" the power consumption profile (i.e. making the power consumption invariant for all possible values of the secret key) or to randomize the power consumption so that a profile can no longer be correlated to any secret information. Masking, on the other hand, conceals every key-dependent intermediate result with a random value, the so-called mask, in order to break the dependency between the sensitive variable (*i.e.* involving the secret key) and the power consumption.

The masking countermeasure. Though masking is often considered to be less efficient (in terms of execution time) than hiding, it provides the key benefit that one can formally prove its security under certain assumptions on the device leakage model and the attacker's capabilities. The way masking is applied depends on the concrete operations executed by a cipher. In general, logical operations (e.g. XOR, Shift, etc.) are protected using Boolean masking, whereas additions/subtractions and multiplications require arithmetic and multiplicative masking, respectively. When a cryptographic algorithm involves a combination of these operations, it becomes necessary to convert the masks from one form to the other in order to get the correct result. Examples of algorithms that perform both arithmetic (e.g. modular addition) and logical operations include two SHA-3 finalists (namely Blake and Skein) as well as all four stream ciphers in the eSTREAM software portfolio. Also, ARX-based block ciphers (e.g. XTEA [NW97] and Threefish) and the hash functions SHA-1 and SHA-2 fall into this category. From a design point of view, modular addition gives essential non-linearity with increased throughput and hence is used in several lightweight block ciphers e.g. SPECK [BSS+13]. Therefore, techniques for conversion between Boolean and arithmetic masks are of significant practical importance.

Conversion between Boolean and arithmetic masking. At CHES 2001, Goubin described a very elegant algorithm for converting from Boolean masking to arithmetic masking, with only a *constant* number of operations, independent of the addition bit size k. Goubin also described an algorithm for converting from arithmetic to Boolean masking, but with $\mathcal{O}(k)$ operations. A different arithmetic to Boolean conversion algorithm was later described in [CT03], based on precomputed tables; an extension was described in [NP04] to reduce the memory consumption. At CHES 2012, Debraize described a modification of the table-based conversion in [CT03], correcting a bug and improving time performances, still with asymptotic complexity $\mathcal{O}(k)$.

Karroumi *et al.* recently noticed in [KRJ14] that Goubin's recursion formula for converting from arithmetic to Boolean masking can also be used to compute an arithmetic addition $z = x+y \bmod 2^k$ directly with masked shares $x = x_1 \oplus x_2$ and $y = y_1 \oplus y_2$. The advantage of this method is that one doesn't need to follow the three step process, i.e. converting x and y from Boolean to arithmetic masking, then performing the addition with arithmetic masks and then converting back from arithmetic to Boolean masks. The authors showed that this can lead to better performances in practice for the block cipher XTEA. However, as their algorithm is based on Goubin's recursion formula, its complexity is still $\mathcal{O}(k)$.

Conversion algorithms have recently been extended to higher-order countermeasure in [CGV14], based on Goubin's conversion method. For security against any attack of order t, their solution has time complexity $\mathcal{O}(n^2 \cdot k)$ for $n = 2t + 1$ shares.

New algorithms with logarithmic complexity. In this paper we describe a new algorithm for converting from arithmetic to Boolean masking with complexity $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$. Our algorithm is based on the Kogge-Stone carry look-ahead adder [KS73], which computes the carry signal in $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$ for the classical ripple carry adder. Following [BN05] and [KRJ14] we also describe a variant algorithm for performing arithmetic addition modulo 2^k directly on Boolean shares, with complexity $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$. We prove the security of our new algorithms against first-order attacks. Our improved solution naturally extends to higher-order masking and reduces the time complexity for n shares from $\mathcal{O}(n^2 \cdot k)$ in [CGV14] down to $\mathcal{O}(n^2 \log k)$.

We also provide implementation results for our algorithms along with existing algorithms on a 32-bit microcontroller. Our results show that the new algorithms perform better than Goubin's algorithm for $k \geq 32$, as we obtain 14% improvement in execution time for $k = 32$, and 23% improvement for $k = 64$. We also describe our results for first-order secure implementations of HMAC-SHA-1 ($k = 32$) and of the SPECK block-cipher ($k = 64$).

2 Goubin's Algorithms

In this section we first recall Goubin's algorithm for converting from Boolean masking to arithmetic masking and conversely [Gou01], secure against first-order attacks. Given a k-bit variable x, for Boolean masking we write:

$$x = x' \oplus r$$

where x' is the masked variable and $r \leftarrow \{0,1\}^k$. Similarly for arithmetic masking we write

$$x = A + r \bmod 2^k$$

In the following all additions and subtractions are done modulo 2^k, for some parameter k.

The goal of the paper is to describe efficient conversion algorithms between Boolean and arithmetic masking, secure against first-order attacks. Given x' and r, one should compute the arithmetic mask $A = (x' \oplus r) - r \bmod 2^k$ without leaking information about $x = x' \oplus r$; this implies that one cannot compute $A = (x' \oplus r) - r \bmod 2^k$ directly, as this would leak information about the sensitive variable $x = x' \oplus r$; instead all intermediate variables should be properly randomized so that no information is leaked about x. Similarly given A and r one must compute the Boolean mask $x' = (A + r) \oplus r$ without leaking information about $x = A + r$.

2.1 Boolean to Arithmetic Conversion

We first recall the Boolean to arithmetic conversion algorithm from Goubin [Gou01]. One considers the following function $\Psi_{x'}(r) : \mathbb{F}_{2^k} \rightarrow \mathbb{F}_{2^k}$:

$$\Psi_{x'}(r) = (x' \oplus r) - r$$

Theorem 1 (Goubin [Gou01]). *The function $\Psi_{x'}(r) = (x' \oplus r) - r$ is affine over $\mathbb{F}_2$.*

Using this affine property, the conversion from Boolean to arithmetic masking is straightforward. Given $x', r \in \mathbb{F}_{2^k}$ we must compute A such that $x' \oplus r = A + r$. From the affine property of $\Psi_{x'}(r)$ we can write:

$$A = (x' \oplus r) - r = \Psi_{x'}(r) = \Psi_{x'}(r \oplus r_2) \oplus \big(\Psi_{x'}(r_2) \oplus \Psi_{x'}(0)\big)$$

for any $r_2 \in \mathbb{F}_{2^k}$. Therefore the technique consists in first generating a uniformly distributed random r_2 in $\mathbb{F}_{2^k}$, then computing $\Psi_{x'}(r \oplus r_2)$ and $\Psi_{x'}(r_2) \oplus \Psi_{x'}(0)$ separately, and finally performing XOR operation on these two to get A. The technique is clearly secure against first-order attacks; namely the left term $\Psi_{x'}(r \oplus r_2)$ is independent from r and therefore from $x = x' \oplus r$, and the right term $\Psi_{x'}(r_2) \oplus \Psi_{x'}(0)$ is also independent from r and therefore from x. Note that the technique is very efficient as it requires only a constant number of operations (independent of k).

2.2 From Arithmetic to Boolean Masking

Goubin also described in [Gou01] a technique for converting from arithmetic to Boolean masking, secure against first-order attacks. However it is more complex than from Boolean to arithmetic masking; its complexity is $\mathcal{O}(k)$ for additions modulo 2^k. It is based on the following theorem.

Theorem 2 (Goubin [Gou01]). *If we denote $x' = (A + r) \oplus r$, we also have $x' = A \oplus u_{k-1}$, where u_{k-1} is obtained from the following recursion formula:*

$$\begin{cases} u_0 = 0 \\ \forall k \geq 0, u_{k+1} = 2[u_k \wedge (A \oplus r) \oplus (A \wedge r)] \end{cases} \tag{1}$$

Since the iterative computation of u_i contains only XOR and AND operations, it can easily be protected against first-order attacks. We refer to Appendix A for the full conversion algorithm.

3 A New Recursive Formula based on Kogge-Stone Adder

Our new conversion algorithm is based on the Kogge-Stone adder [KS73], a carry look-ahead adder that generates the carry signal in $\mathcal{O}(\log k)$ time, when addition is performed modulo 2^k. In this section we first recall the classical ripple-carry adder, which generates the carry signal in $\mathcal{O}(k)$ time, and we show how Goubin's recursion formula (1) can be derived from it. The derivation of our new recursion formula from the Kogge-Stone adder will proceed similarly.

3.1 The Ripple-Carry Adder and Goubin's Recursion Formula

We first recall the classical ripple-carry adder. Given three bits x, y and c, the carry c' for $x+y+c$ can be computed as $c' = (x \wedge y) \oplus (x \wedge c) \oplus (y \wedge c)$. Therefore, the modular addition of two k-bit variables x and y can be defined recursively as follows:

$$(x + y)^{(i)} = x^{(i)} \oplus y^{(i)} \oplus c^{(i)} \tag{2}$$

for $0 \leq i < k$, where

$$\begin{cases} c^{(0)} = 0 \\ \forall i \geq 1,\ c^{(i)} = (x^{(i-1)} \wedge y^{(i-1)}) \oplus (x^{(i-1)} \wedge c^{(i-1)}) \oplus (c^{(i-1)} \wedge y^{(i-1)}) \end{cases} \tag{3}$$

where $x^{(i)}$ represents the i^{th} bit of the variable x, with $x^{(0)}$ being the least significant bit.

In the following, we show how recursion (3) can be computed directly with k-bit values instead of bits, which enables us to recover Goubin's recursion (1). For this, we define the sequences x_j, y_j and v_j whose $j+1$ least significant bits are the same as x, y and c respectively:

$$x_j = \bigoplus_{i=0}^{j} 2^i x^{(i)}, \quad y_j = \bigoplus_{i=0}^{j} 2^i y^{(i)}, \quad v_j = \bigoplus_{i=0}^{j} 2^i c^{(i)} \tag{4}$$

for $0 \leq j \leq k-1$. Since $c^{(0)} = 0$ we can actually start the summation for v_j at $i = 1$; we get from (3):

$$\begin{aligned}
v_{j+1} &= \bigoplus_{i=1}^{j+1} 2^i c^{(i)} \\
v_{j+1} &= \bigoplus_{i=1}^{j+1} 2^i \left((x^{(i-1)} \wedge y^{(i-1)}) \oplus (x^{(i-1)} \wedge c^{(i-1)}) \oplus (c^{(i-1)} \wedge y^{(i-1)}) \right) \\
v_{j+1} &= 2 \bigoplus_{i=0}^{j} 2^i \left((x^{(i)} \wedge y^{(i)}) \oplus (x^{(i)} \wedge c^{(i)}) \oplus (c^{(i)} \wedge y^{(i)}) \right) \\
v_{j+1} &= 2\big((x_j \wedge y_j) \oplus (x_j \wedge v_j) \oplus (y_j \wedge v_j) \big)
\end{aligned}$$

which gives the recursive equation:

$$\begin{cases} v_0 = 0 \\ \forall j \geq 0,\ v_{j+1} = 2\left(v_j \wedge (x_j \oplus y_j) \oplus (x_j \wedge y_j)\right) \end{cases} \tag{5}$$

Therefore we have obtained a recursion similar to (3), but with k-bit values instead of single bits. Note that from the definition of v_j in (4) the variables v_j and v_{j+1} have the same least significant bits from bit 0 to bit j, which is not immediately obvious when considering only recursion (5). Combining (2) and (4) we obtain $x_j + y_j = x_j \oplus y_j \oplus v_j$ for all $0 \leq j \leq k-1$. For k-bit values x and y, we have $x = x_{k-1}$ and $y = y_{k-1}$, which gives:

$$x + y = x \oplus y \oplus v_{k-1}$$

We now define the same recursion as (5), but with constant x, y instead of x_j, y_j. That is, we let

$$\begin{cases} u_0 = 0 \\ \forall j \geq 0,\ u_{j+1} = 2\left(u_j \wedge (x \oplus y) \oplus (x \wedge y)\right) \end{cases} \tag{6}$$

which is exactly the same recursion as Goubin's recursion (1). It is easy to show inductively that the variables u_j and v_j have the same least significant bits, from bit 0 to bit j. Let us assume that this is true for u_j and v_j. From recursions (5) and (6) we have that the least significant bits of v_{j+1} and u_{j+1} from bit 0 to bit $j+1$ only depend on the least significant bits from bit 0 to bit j of v_j, x_j and y_j, and of u_j, x and y respectively. Since these are the same, the induction is proved.

Eventually for k-bit registers we have $u_{k-1} = v_{k-1}$, which proves Goubin's recursion formula (1), namely:

$$x + y = x \oplus y \oplus u_{k-1}$$

As mentioned previously, this recursion formula requires $k-1$ iterations on k-bit registers. In the following, we describe an improved recursion based on the Kogge-Stone carry look-ahead adder, requiring only $\log_2 k$ iterations.

3.2 The Kogge-Stone Carry Look-Ahead Adder

In this section we first recall the general solution from [KS73] for first-order recurrence equations; the Kogge-Stone carry look-ahead adder is a direct application.

General first-order recurrence equation. We consider the following recurrence equation:

$$\begin{cases} z_0 = b_0 \\ \forall i \geq 1,\ z_i = a_i z_{i-1} + b_i \end{cases} \tag{7}$$

We define the function $Q(m, n)$ for $m \geq n$:

$$Q(m,n) = \sum_{j=n}^{m} \left(\prod_{i=j+1}^{m} a_i \right) b_j \tag{8}$$

We have $Q(0,0) = b_0 = z_0$, $Q(1,0) = a_1 b_0 + b_1 = z_1$, and more generally:

$$\begin{aligned} Q(m,0) &= \sum_{j=0}^{m-1} \left(\prod_{i=j+1}^{m} a_i \right) b_j + b_m \\ &= a_m \sum_{j=0}^{m-1} \left(\prod_{i=j+1}^{m-1} a_i \right) b_j + b_m = a_m Q(m-1,0) + b_m \end{aligned}$$

Therefore the sequence $Q(m,0)$ satisfies the same recurrence as z_m, which implies $Q(m,0) = z_m$ for all $m \geq 0$. Moreover we have:

$$\begin{aligned} Q(2m-1,0) &= \sum_{j=0}^{2m-1} \left(\prod_{i=j+1}^{2m-1} a_i \right) b_j \\ &= \left(\prod_{j=m}^{2m-1} a_j \right) \sum_{j=0}^{m-1} \left(\prod_{i=j+1}^{m-1} a_i \right) b_j + \sum_{j=m}^{2m-1} \left(\prod_{i=j+1}^{2m-1} a_i \right) b_j \end{aligned}$$

which gives the recursive doubling equation:

$$Q(2m-1,0) = \left(\prod_{j=m}^{2m-1} a_j \right) Q(m-1,0) + Q(2m-1,m)$$

where each term $Q(m-1,0)$ and $Q(2m-1,m)$ contain only m terms a_i and b_i, instead of $2m$ in $Q(2m-1,0)$. Therefore the two terms can be computed in parallel. This is also the case for the product $\prod_{j=m}^{2m-1} a_j$ which can be computed with a product tree. Therefore by recursive splitting with N processors, the sequence element z_N can be computed in time $\mathcal{O}(\log_2 N)$, instead of $\mathcal{O}(N)$ with a single processor.

The Kogge-Stone Carry Look-Ahead Adder. The Kogge-Stone carry look-ahead adder [KS73] is a direct application of the previous technique. Namely writing $c_i = c^{(i)}$, $a_i = x^{(i)} \oplus y^{(i)}$ and $b_i = x^{(i)} \wedge y^{(i)}$ for all $i \geq 0$, we obtain from (3) the recurrence relation for the carry signal c_i:

$$\begin{cases} c_0 = 0 \\ \forall i \geq 1, \;\; c_i = (a_{i-1} \wedge c_{i-1}) \oplus b_{i-1} \end{cases}$$

which is similar to (7), where $\wedge$ is the multiplication and $\oplus$ the addition. We can therefore compute the carry signal c_i for $0 \leq i < k$ in time $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$.

More precisely, the Kogge-Stone carry look-ahead adder can be defined as follows. For all $0 \leq j < k$ one defines the sequence of bits:

$$P_{0,j} = x^{(j)} \oplus y^{(j)}, \quad G_{0,j} = x^{(j)} \wedge y^{(j)} \tag{9}$$

and the following recursive equations:

$$\begin{cases} P_{i,j} = P_{i-1,j} \wedge P_{i-1,j-2^{i-1}} \\ G_{i,j} = (P_{i-1,j} \wedge G_{i-1,j-2^{i-1}}) \oplus G_{i-1,j} \end{cases} \tag{10}$$

for $2^{i-1} \leq j < k$, and $P_{i,j} = P_{i-1,j}$ and $G_{i,j} = G_{i-1,j}$ for $0 \leq j < 2^{i-1}$. The following lemma shows that the carry signal c_j can be computed from the sequence $G_{i,j}$.

Lemma 1. *We have $(x+y)^{(j)} = x^{(j)} \oplus y^{(j)} \oplus c_j$ for all $0 \leq j < k$ where the carry signal c_j is computed as $c_0 = 0$, $c_1 = G_{0,0}$ and $c_{j+1} = G_{i,j}$ for $2^{i-1} \leq j < 2^i$.*

To compute the carry signal up to c_{k-1}, one must therefore compute the sequences $P_{i,j}$ and $G_{i,j}$ up to $i = \lceil \log_2(k-1) \rceil$. For completeness we provide the proof of Lemma 1 in Appendix B.

3.3 Our New Recursive Algorithm

We now derive a recursion formula with k-bit variables instead of single bits; we proceed as in Section 3.1, using the more efficient Kogge-Stone carry look-ahead algorithm, instead of the classical ripple-carry adder for Goubin's recursion. We prove the following theorem, analogous to Theorem 2, but with complexity $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$. Given a variable x, we denote by $x \ll \ell$ the variable x left-shifted by ℓ bits, keeping only k bits in total.

Theorem 3. *Let $x, y \in \{0,1\}^k$ and $n = \lceil \log_2(k-1) \rceil$. Define the sequence of k-bit variables P_i and G_i, with $P_0 = x \oplus y$ and $G_0 = x \wedge y$, and*

$$\begin{cases} P_i = P_{i-1} \wedge (P_{i-1} \lll 2^{i-1}) \\ G_i = \left(P_{i-1} \wedge (G_{i-1} \lll 2^{i-1})\right) \oplus G_{i-1} \end{cases} \tag{11}$$

for $1 \le i \le n$. Then $x + y = x \oplus y \oplus (2G_n)$.

Proof. We start from the sequences $P_{i,j}$ and $G_{i,j}$ defined in Section 3.2 corresponding to the Kogge-Stone carry look-ahead adder, and we proceed as in Section 3.1. We define the variables:

$$P_i := \sum_{j=2^i-1}^{k-1} 2^j P_{i,j} \quad G_i := \sum_{j=0}^{k-1} 2^j G_{i,j}$$

which from (9) gives the initial condition $P_0 = x \oplus y$ and $G_0 = x \wedge y$, and using (10):

$$\begin{aligned} P_i &= \sum_{j=2^i-1}^{k-1} 2^j P_{i,j} = \sum_{j=2^i-1}^{k-1} 2^j (P_{i-1,j} \wedge P_{i-1,j-2^{i-1}}) \\ &= \left(\sum_{j=2^i-1}^{k-1} 2^j P_{i-1,j} \right) \wedge \left(\sum_{j=2^i-1}^{k-1} 2^j P_{i-1,j-2^{i-1}} \right) \end{aligned}$$

We can start the summation of the $P_{i,j}$ bits with $j = 2^{i-1} - 1$ instead of $2^i - 1$, because the other summation still starts with $j = 2^i - 1$, hence the corresponding bits are ANDed with 0. This gives:

$$\begin{aligned} P_i &= \left(\sum_{j=2^{i-1}-1}^{k-1} 2^j P_{i-1,j} \right) \wedge \left(\sum_{j=2^i-1}^{k-1} 2^j P_{i-1,j-2^{i-1}} \right) \\ &= P_{i-1} \wedge \left(\sum_{j=2^{i-1}-1}^{k-1-2^{i-1}} 2^{j+2^{i-1}} P_{i-1,j} \right) = P_{i-1} \wedge (P_{i-1} \lll 2^{i-1}) \end{aligned}$$

Hence we get the same recursion formula for P_i as in (11). Similarly we have using (10):

$$\begin{aligned} G_i = \sum_{j=0}^{k-1} 2^j G_{i,j} &= \sum_{j=2^{i-1}}^{k-1} 2^j \left((P_{i-1,j} \wedge G_{i-1,j-2^{i-1}}) \oplus G_{i-1,j} \right) + \sum_{j=0}^{2^{i-1}-1} 2^j G_{i-1,j} \\ &= \left(\sum_{j=2^{i-1}}^{k-1} 2^j \left(P_{i-1,j} \wedge G_{i-1,j-2^{i-1}} \right) \right) \oplus G_{i-1} \\ &= \left(P_{i-1} \wedge (G_{i-1} \lll 2^{i-1}) \right) \oplus G_{i-1} \end{aligned}$$

Therefore we obtain the same recurrence for P_i and G_i as (11). Since from Lemma 1 we have that $c_{j+1} = G_{i,j}$ for all $2^{i-1} \le j < 2^i$, and $G_{i,j} = G_{i-1,j}$ for $0 \le j < 2^{i-1}$, we obtain $c_{j+1} = G_{i,j}$ for all $0 \le j < 2^i$. Taking $i = n = \lceil \log_2(k-1) \rceil$, we obtain $c_{j+1} = G_{n,j}$ for all $0 \le j \le k-2 < k-1 \le 2^n$. This implies:

$$\sum_{j=0}^{k-1} 2^j c_j = \sum_{j=1}^{k-1} 2^j c_j = 2 \sum_{j=0}^{k-2} 2^j c_{j+1} = 2 \sum_{j=0}^{k-2} 2^j G_{n,j} = 2G_n$$

Since from Lemma 1 we have $(x+y)^{(j)} = x^{(j)} \oplus y^{(j)} \oplus c_j$ for all $0 \le j < k$, this implies $x + y = x \oplus y \oplus (2G_n)$ as required. □

The complexity of the previous recursion is only $\mathcal{O}(\log k)$, as opposed to $\mathcal{O}(k)$ with Goubin's recursion. The sequence can be computed using the algorithm below; note that we do not compute the last element P_n since it is not used in the computation of G_n. Note also that the algorithm below could be used as a $\mathcal{O}(\log k)$ implementation of arithmetic addition $z = x + y \bmod 2^k$ for processors having only Boolean operations.

Algorithm 1 Kogge-Stone Adder

```
Input: $x, y \in \{0,1\}^k$, and $n = \max(\lceil \log_2(k-1) \rceil, 1)$.
Output: $z = x + y \bmod 2^k$
 1: $P \leftarrow x \oplus y$
 2: $G \leftarrow x \wedge y$
 3: for $i := 1$ to $n - 1$ do
 4:     $G \leftarrow (P \wedge (G \ll 2^{i-1})) \oplus G$
 5:     $P \leftarrow P \wedge (P \ll 2^{i-1})$
 6: end for
 7: $G \leftarrow (P \wedge (G \ll 2^{n-1})) \oplus G$
 8: return $x \oplus y \oplus (2G)$
```

4 Our New Conversion Algorithm

Our new conversion algorithm from arithmetic to Boolean masking is a direct application of the Kogge-Stone adder in Algorithm 1. We are given as input two arithmetic shares A, r of $x = A + r \bmod 2^k$, and we must compute x' such that $x = x' \oplus r$, without leaking information about x.

Since Algorithm 1 only contains Boolean operations, it is easy to protect against first-order attacks. Assume that we give as input the two arithmetic shares A and r to Algorithm 1; the algorithm first computes $P = A \oplus r$ and $G = A \wedge r$, and after n iterations outputs $x = A + r = A \oplus r \oplus (2G)$. Obviously one cannot compute $P = A \oplus r$ and $G = A \wedge r$ directly since that would reveal information about the sensitive variable $x = A + r$. Instead we protect all intermediate variables with a random mask s using standard techniques, that is we only work with $P' = P \oplus s$ and $G' = G \oplus s$. Eventually we obtain a masked $x' = x \oplus s$ as required, in time $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$.

4.1 Secure Computation of AND

Since Algorithm 1 contains AND operations, we first show how to secure the AND operation against first-order attacks. The technique is essentially the same as in [ISW03]. With $x = x' \oplus s$ and $y = y' \oplus t$ for two independent random masks s and t, we have for any u:

$$(x \wedge y) \oplus u = \big((x' \oplus s) \wedge (y' \oplus t)\big) \oplus u = (x' \wedge y') \oplus (x' \wedge t) \oplus (s \wedge y') \oplus (s \wedge t) \oplus u$$

Algorithm 2 SecAnd

```
Input: $x'$, $y'$, $s$, $t$, $u$ such that $x' = x \oplus s$ and $y' = y \oplus t$.
Output: $z'$ such that $z' = (x \wedge y) \oplus u$.
 1: $z' \leftarrow u \oplus (x' \wedge y')$
 2: $z' \leftarrow z' \oplus (x' \wedge t)$
 3: $z' \leftarrow z' \oplus (s \wedge y')$
 4: $z' \leftarrow z' \oplus (s \wedge t)$
 5: return $z'$
```

We see that the SecAnd algorithm requires 8 Boolean operations. The following Lemma shows that the SecAnd algorithm is secure against first-order attacks.

Lemma 2. *When s, t and u are uniformly and independently distributed in $\mathbb{F}_{2^k}$, all intermediate variables in the* SecAnd *algorithm have a distribution independent from x and y.*

Proof. Since s and t are uniformly and independently distributed in $\mathbb{F}_{2^k}$, the variables $x' = x \oplus s$ and $y' = y \oplus t$ are also uniformly and independently distributed in $\mathbb{F}_{2^k}$. Therefore the distribution of $x' \wedge y'$ is independent from x and y. The same holds for the variables $x' \wedge t$, $s \wedge y'$ and $s \wedge t$. Moreover since u is uniformly distributed in $\mathbb{F}_{2^k}$, the distribution of z' from Line 1 to Line 4 is uniform in $\mathbb{F}_{2^k}$; hence its distribution is also independent from x and y. □

4.2 Secure Computation of XOR

Similarly we show how to secure the XOR computation of Algorithm 1. With $x = x' \oplus s$ and $y = y' \oplus u$ where s and u are two independent masks, we have:

$$(x \oplus y) \oplus s = x' \oplus s \oplus y' \oplus u \oplus s = x' \oplus y' \oplus u$$

Algorithm 3 SecXor

```
Input: $x'$, $y'$, $u$, such that $x' = x \oplus s$, and $y' = y \oplus u$.
Output: $z'$ such that $z' = (x \oplus y) \oplus s$.
$z' \leftarrow x' \oplus y'$
$z' \leftarrow z' \oplus u$
return $z'$
```

We see that the SecXor algorithm requires 2 Boolean operations. The following Lemma shows that the SecXor algorithm is secure against first-order attacks. It is easy to see that all the intermediate variables in the algorithm are uniformly distributed in $\mathbb{F}_{2^k}$, and hence the proof is straightforward.

Lemma 3. *When s and u are uniformly and independently distributed in $\mathbb{F}_{2^k}$, all intermediate variables in the* SecXor *algorithm have a distribution independent from x and y.*

4.3 Secure Computation of Shift

Finally we show how to secure the Shift operation in Algorithm 1 against first-order attacks. With $x = x' \oplus s$, we have for any t:

$$(x \ll j) \oplus t = \big((x' \oplus s) \ll j\big) \oplus t = (x' \ll j) \oplus (s \ll j) \oplus t$$

This gives the following algorithm.

Algorithm 4 SecShift

```
Input: $x'$, $s$, $t$ and $j$ such that $x' = x \oplus s$ and $j > 0$.
Output: $y'$ such that $y' = (x \ll j) \oplus t$.
$y' \leftarrow t \oplus (x' \ll j)$
$y' \leftarrow y' \oplus (s \ll j)$
return $y'$
```

We see that the SecShift algorithm requires 4 Boolean operations. The following Lemma shows that the SecShift algorithm is secure against first-order attacks. The proof is straightforward so we omit it.

Lemma 4. *When s and t are uniformly and independently distributed in $\mathbb{F}_{2^k}$, all intermediate variables in the* SecShift *algorithm have a distribution independent from x.*

4.4 Our New Conversion Algorithm

Finally we can convert Algorithm 1 into a first-order secure algorithm by protecting all intermediate variables with a random mask; see Algorithm 5 below.

Since the SecAnd subroutine requires 8 operations, the SecXor subroutine requires 2 operations, and the SecShift subroutine requires 4 operations, lines 7 to 11 require $2 \cdot 8 + 2 \cdot 4 + 2 + 2 = 28$ operations, hence $28 \cdot (n-1)$ operations for the main loop. The total number of operations is then $7 + 28 \cdot (n-1) + 4 + 8 + 2 + 4 = 28 \cdot n - 3$. In summary, for a register size $k = 2^n$ the number of operations is $28 \cdot \log_2 k - 3$, in addition to the generation of 3 random numbers. Note that the same random numbers s, t and u can actually be used for all executions of the conversion algorithm in a given execution. The following Lemma proves the security of our new conversion algorithm against first-order attacks.

Lemma 5. *When r is uniformly distributed in $\mathbb{F}_{2^k}$, any intermediate variable in Algorithm 5 has a distribution independent from $x = A + r \bmod 2^k$.*

Proof. The proof is based on the previous lemma for SecAnd, SecXor and SecShift, and also the fact that all intermediate variables from Line 2 to 5 and in lines 12, 13, 18, and 19 have a distribution independent from x. Namely $(A \oplus t) \wedge r$ and $t \wedge r$ have a distribution independent from x, and the other intermediate variables have the uniform distribution. □

Algorithm 5 Kogge-Stone Arithmetic to Boolean Conversion

```
Input: $A, r \in \{0,1\}^k$ and $n = \max(\lceil \log_2(k-1) \rceil, 1)$
Output: $x'$ such that $x' \oplus r = A + r \bmod 2^k$.
 1: Let $s \leftarrow \{0,1\}^k$, $t \leftarrow \{0,1\}^k$, $u \leftarrow \{0,1\}^k$.
 2: $P' \leftarrow A \oplus s$
 3: $P' \leftarrow P' \oplus r$                              ▷ $P' = (A \oplus r) \oplus s = P \oplus s$
 4: $G' \leftarrow s \oplus \big((A \oplus t) \wedge r\big)$
 5: $G' \leftarrow G' \oplus (t \wedge r)$                  ▷ $G' = (A \wedge r) \oplus s = G \oplus s$
 6: for $i := 1$ to $n-1$ do
 7:     $H \leftarrow \mathsf{SecShift}(G', s, t, 2^{i-1})$        ▷ $H = (G \ll 2^{i-1}) \oplus t$
 8:     $U \leftarrow \mathsf{SecAnd}(P', H, s, t, u)$           ▷ $U = \big(P \wedge (G \ll 2^{i-1})\big) \oplus u$
 9:     $G' \leftarrow \mathsf{SecXor}(G', U, u)$               ▷ $G' = \big((P \wedge (G \ll 2^{i-1})) \oplus G\big) \oplus s$
10:     $H \leftarrow \mathsf{SecShift}(P', s, t, 2^{i-1})$        ▷ $H = (P \ll 2^{i-1}) \oplus t$
11:     $P' \leftarrow \mathsf{SecAnd}(P', H, s, t, u)$          ▷ $P' = \big(P \wedge (P \ll 2^{i-1})\big) \oplus u$
12:     $P' \leftarrow P' \oplus s$
13:     $P' \leftarrow P' \oplus u$                          ▷ $P' = \big(P \wedge (P \ll 2^{i-1})\big) \oplus s$
14: end for
15: $H \leftarrow \mathsf{SecShift}(G', s, t, 2^{n-1})$            ▷ $H = (G \ll 2^{n-1}) \oplus t$
16: $U \leftarrow \mathsf{SecAnd}(P', H, s, t, u)$               ▷ $U = \big(P \wedge (G \ll 2^{n-1})\big) \oplus u$
17: $G' \leftarrow \mathsf{SecXor}(G', U, u)$                   ▷ $G' = \big((P \wedge (G \ll 2^{n-1})) \oplus G\big) \oplus s$
18: $x' \leftarrow A \oplus 2G'$                              ▷ $x' = (A + r) \oplus r \oplus 2s$
19: $x' \leftarrow x' \oplus 2s$                              ▷ $x' = (A + r) \oplus r$
20: return $x'$
```

5 Addition Without Conversion

Beak and Noh proposed a method to mask the ripple carry adder in [BN05]. Similarly, Karroumi *et al.* [KRJ14] used Goubin's recursion formula (1) to compute an arithmetic addition $z = x+y \bmod 2^k$ directly with masked shares $x' = x \oplus s$ and $y' = y \oplus r$, that is without first converting x and y from Boolean to arithmetic masking, then performing the addition with arithmetic masks, and then converting back from arithmetic to Boolean masks. They showed that this can lead to better performances in practice for the block cipher XTEA.

In this section we describe an analogous algorithm for performing addition directly on the masked shares, based on the Kogge-Stone adder instead of Goubin's formula, to get $\mathcal{O}(\log k)$

complexity instead of $\mathcal{O}(k)$. More precisely, we receive as input the shares x', y' such that $x' = x \oplus s$ and $y' = y \oplus r$, and the goal is to compute z' such that $z' = (x + y) \oplus r$. For this it suffices to perform the addition $z = x + y \bmod 2^k$ as in Algorithm 1, but with the masked variables $x' = x \oplus s$ and $y' = y \oplus r$ instead of x, y, while protecting all intermediate variables with a Boolean mask; this is straightforward since Algorithm 1 contains only Boolean operations; see Algorithm 6 below.

Algorithm 6 Kogge-Stone Masked Addition

```
Input: $x', y', r, s \in \{0,1\}^k$ and $n = \max(\lceil \log_2(k-1) \rceil, 1)$.
Output: $z'$ such that $z' = (x+y) \oplus r$, where $x = x' \oplus s$ and $y = y' \oplus r$
Let $t \leftarrow \{0,1\}^k$, $u \leftarrow \{0,1\}^k$.
$P' \leftarrow \mathsf{SecXor}(x', y', r)$                          ▷ $P' = (x \oplus y) \oplus s = P \oplus s$
$G' \leftarrow \mathsf{SecAnd}(x', y', s, r, u)$                     ▷ $G' = (x \wedge y) \oplus u = G \oplus u$
$G' \leftarrow G' \oplus s$
$G' \leftarrow G' \oplus u$                                          ▷ $G' = (x \wedge y) \oplus s = G \oplus s$
for $i := 1$ to $n-1$ do
    $H \leftarrow \mathsf{SecShift}(G', s, t, 2^{i-1})$              ▷ $H = (G \ll 2^{i-1}) \oplus t$
    $U \leftarrow \mathsf{SecAnd}(P', H, s, t, u)$                   ▷ $U = \big(P \wedge (G \ll 2^{i-1})\big) \oplus u$
    $G' \leftarrow \mathsf{SecXor}(G', U, u)$                        ▷ $G' = \big((P \wedge (G \ll 2^{i-1})) \oplus G\big) \oplus s$
    $H \leftarrow \mathsf{SecShift}(P', s, t, 2^{i-1})$              ▷ $H = (P \ll 2^{i-1}) \oplus t$
    $P' \leftarrow \mathsf{SecAnd}(P', H, s, t, u)$                  ▷ $P' = \big(P \wedge (P \ll 2^{i-1})\big) \oplus u$
    $P' \leftarrow P' \oplus s$
    $P' \leftarrow P' \oplus u$                                      ▷ $P' = \big(P \wedge (P \ll 2^{i-1})\big) \oplus s$
end for
$H \leftarrow \mathsf{SecShift}(G', s, t, 2^{n-1})$                  ▷ $H = (G \ll 2^{n-1}) \oplus t$
$U \leftarrow \mathsf{SecAnd}(P', H, s, t, u)$                       ▷ $U = \big(P \wedge (G \ll 2^{n-1})\big) \oplus u$
$G' \leftarrow \mathsf{SecXor}(G', U, u)$                            ▷ $G' = \big((P \wedge (G \ll 2^{n-1})) \oplus G\big) \oplus s$
$z' \leftarrow \mathsf{SecXor}(y', x', s)$                           ▷ $z' = (x \oplus y) \oplus r$
$z' \leftarrow z' \oplus (2G')$                                      ▷ $z' = (x+y) \oplus 2s \oplus r$
$z' \leftarrow z' \oplus 2s$                                         ▷ $z' = (x+y) \oplus r$
return $z'$
```

As previously the main loop requires $28 \cdot (n-1)$ operations. The total number of operations is then $12 + 28 \cdot (n-1) + 20 = 28 \cdot n + 4$. In summary, for a register size $k = 2^n$ the number of operations is $28 \cdot \log_2 k + 4$, with additionally the generation of 2 random numbers; as previously those 2 random numbers along with r and s can be reused for subsequent additions within the same execution. The following Lemma proves the security of Algorithm 6 against first-order attacks. The proof is similar to the proof of Lemma 5 and is therefore omitted.

Lemma 6. *For a uniformly and independently distributed randoms $r \in \{0,1\}^k$ and $s \in \{0,1\}^k$, any intermediate variable in the* Kogge-Stone Masked Addition *has the uniform distribution.*

6 Extension to Higher-Order Masking

The first conversion algorithms between Boolean and arithmetic masking secure against t-th order attack (instead of first-order only) were presented in [CGV14]. The authors first described an algorithm for secure addition modulo 2^k directly with n Boolean shares (where $n \geq 2t+1$), with complexity $\mathcal{O}(n^2 \cdot k)$. The algorithm was then used as a subroutine to obtain conversion algorithms in both directions, again with complexity $\mathcal{O}(n^2 \cdot k)$. The algorithms were proven secure in the ISW framework for private circuits [ISW03].

Our improved solution is easily adapted to obtain addition modulo 2^k and conversion algorithms with complexity $\mathcal{O}(n^2 \cdot \log k)$ instead of $\mathcal{O}(n^2 \cdot k)$. Namely within [CGV14] it suffices to replace the $\mathcal{O}(k)$ Goubin's addition from Theorem 2 by the $\mathcal{O}(\log k)$ Kogge-Stone addition from Theorem 3. The resulting algorithms are still proven secure in the ISW framework.

7 Analysis and Implementation

7.1 Comparison With Existing Algorithms

We compare in Table 1 the complexity of our new algorithms with Goubin's algorithms and Debraize's algorithms for various addition bit sizes k.[1] We give the number of random numbers required for each of the algorithms as well the number of elementary operations. Goubin's original conversion algorithm from arithmetic to Boolean masking required $5k + 5$ operations and a single random generation. This was recently improved by Karroumi *et al.* down to $5k + 1$ operations [KRJ14]. The authors also provided an algorithm to compute first-order secure addition on Boolean shares using Goubin's recursion formula, requiring $5k + 8$ operations and a single random generation. See Appendix A for more details. On the other hand Debraize's algorithm requires $19(k/\ell) - 2$ operations with a lookup table of size 2^ℓ and the generation of two randoms.

Algorithm	rand	$k = 8$	$k = 16$	$k = 32$	$k = 64$	k
Goubin's A→						

hand, Debraize's algorithm performs better than our algorithms for $\ell = 8$; however as opposed to Debraize's algorithm our conversion algorithm requires neither preprocessing nor extra memory.

	$k = 8$	$k = 16$	$k = 32$	$k = 64$	Prep.	Mem.
Goubin's A→B conversion	180	312	543	1672	-	-
Debraize's A→B conversion ($\ell = 4$)	149	285	505	1573	1221	32
Debraize's A→B conversion ($\ell = 8$)	-	193	316	846	18024	1024
New A→B conversion	301	386	467	1284	-	-
Goubin's addition [KRJ14]	235	350	582	1789	-	-
New addition	344	429	513	1340	-	-

Table 2. Number of clock cycles on a 32-bit processor required for Goubin's conversion algorithm, Debraize's conversion algorithm, our new conversion algorithm, Goubin's addition from [KRJ14], and our new addition, for various arithmetic sizes k. The last two columns denote the precomputation time and the table size (in bytes) required for Debraize's algorithm.

8 Application to HMAC-SHA-1

In this section, we apply our countermeasure to obtain a first-order secure implementation of HMAC-SHA-1 [NIS95]. Since SHA-1 involves both modular addition and XOR operations, we must either convert between Boolean and arithmetic masking, or perform the arithmetic additions directly on the Boolean shares as suggested in [KRJ14].

8.1 HMAC-SHA-1

SHA-1 processes the input in blocks of 512-bits and produces a message digest of 160 bits. If the length of the message is not a multiple of 512, the message is appended with zeros, followed by a "1". The last 64 bits of the message contains the length of the original message in bits. All operations are performed on 32-bit words, producing an output of five words. We denote by $x \lll i$ the left rotation of the 32-bit word x by i bits. Each 512-bit message block $M[0], \cdots, M[15]$ is expanded to 80 words as follows:

$$W[i] = \begin{cases} M[i] & \text{if } i \leq 15 \\ (W[i-3] \oplus W[i-8] \oplus W[i-14] \oplus W[i-16]) \lll 1 & \text{Otherwise} \end{cases}$$

One initially lets $A = H_0$, $B = H_1$, $C = H_2, D = H_3$ and $E = H_4$, where the H_i's are initial constants. The main loop is defined as follows, for $i = 0$ to $i = 79$.

$$\begin{gathered} \mathit{Temp} = (A \lll 5) + f(i, B, C, D) + E + W[i] + k[i] \\ E = D;\ D = C;\ C = B \lll 30;\ B = A;\ A = \mathit{Temp} \end{gathered}$$

where the function f is defined as:

$$f(i, B, C, D) = \begin{cases} (B \wedge C) \vee ((\neg B) \wedge D) & \text{if } 0 \leq i \leq 19 \\ (B \oplus C \oplus D) & \text{if } 20 \leq i \leq 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & \text{if } 40 \leq i \leq 59 \\ (B \oplus C \oplus D) & \text{if } 60 \leq i \leq 79 \end{cases}$$

and $k[i]$ are constants. After the main loop one lets:

$$H_0 \leftarrow H_0 + A, \quad H_1 \leftarrow H_1 + B, \quad H_2 \leftarrow H_2 + C, \quad H_3 \leftarrow H_3 + C, \quad H_4 \leftarrow H_4 + D \tag{12}$$

and one processes the next block. All additions are performed modulo 2^{32}. Eventually the final hash result is the 160-bit string $H_0\|H_1\|H_2\|H_3\|H_4$.

HMAC-SHA-1 of a message M is computed as:

$$H\big(K \oplus \mathit{opad} \parallel H(K \oplus \mathit{ipad}, M)\big)$$

where H is the SHA-1 function, K is the secret key, and *ipad* and *opad* are constants.

8.2 First-Order Secure HMAC-SHA-1

In this section we show how to protect HMAC-SHA-1 against first-order attacks, using either Goubin's Boolean to arithmetic conversion (Section 2.1) and our new arithmetic to Boolean conversion (Algorithm 5), or the addition with Boolean masks (Algorithm 6).

Since $W[i]$ is a linear function, it is easily protected against first-order attacks. Similarly the function f contains only Boolean operations, so it is easy to protect against side-channel attacks. More precisely, to compute $\neg x$ given the shares of $x = x_1 \oplus x_2$, it suffices to compute $\neg x_1$ since $\neg x = (\neg x_1) \oplus x_2$. The AND operation is computed using SecAnd (Algorithm 2). The OR operation is computed using $x \vee y = \neg((\neg x) \wedge (\neg y))$.

The main loop consists of the following operation:

$$\mathit{Temp} = (A \lll 5) + f(i, B, C, D) + E + W[i] + k[i]$$

If we use the masked addition (Algorithm 6), then we must perform four such additions on the Boolean shares to evaluate the above expression. Since the SHA-1 loop consists of 80 iterations, we need $4 \cdot 80 = 320$ calls to the masked addition (Algorithm 6) for the main loop. Additionally, the update of the H_i's (12) needs five calls. Hence, we need a total of 325 calls per each message block .

Alternatively, one can first convert the Boolean shares to arithmetic shares using Goubin's algorithm (Section 2.1), then perform the addition directly on the arithmetic shares, and finally convert back the resulting arithmetic shares to Boolean shares using Algorithm 5. With this approach we need 5 Boolean to arithmetic conversions and 1 arithmetic to Boolean conversion for each iteration. Hence with the additional update of the H_i's we need a total of $5 \cdot 80 + 5 = 405$ Boolean to arithmetic conversions and $80 + 5 = 85$ arithmetic to Boolean conversions.

8.3 Practical Implementation

We have implemented HMAC-SHA-1 [NIS95] using the technique above to protect against first-order attacks, on the same microcontroller as in Section 7.2. To convert from arithmetic to Boolean masking, we used one of the following: Goubin's algorithm, Debraize's algorithm or our new algorithm. The results for computing HMAC-SHA-1 of a single message block are summarized in Table 3. For Debraize's algorithm, the timings also include the precomputation time required for creating the tables. Our algorithms give better performances than Goubin and Debraize ($\ell = 4$), but Debraize with $\ell = 8$ is still slightly better; however as opposed to Debraize, our algorithms do not require extra memory. For the masked addition (instead of conversions), the new algorithm performs 10% better than Goubin's algorithm.

9 Application to SPECK

SPECK is a family of lightweight block ciphers proposed by NSA, which provides high throughput for application in software [BSS+13]. The SPECK family includes various ciphers based on ARX (Addition, Rotation, XOR) design with different block and key sizes. To verify the performance results of our algorithms for $k = 64$, we used SPECK 128/128, where block and key sizes both equal to 128 and additions are performed modulo 2^{64}. The 128-bit key is expanded to 32 64-bit

	Time	Penalty Factor	Mem.
HMAC-SHA-1 unmasked	128	1	-
HMAC-SHA-1 with Goubin's conversion	423	3.3	-
HMAC-SHA-1 with Debraize's conversion ($\ell = 4$)	418	3.26	32
HMAC-SHA-1 with Debraize's conversion ($\ell = 8$)	402	3.1	1024
HMAC-SHA-1 with new conversion	410	3.2	-
HMAC-SHA-1 with Goubin's addition [KRJ14]	1022	8	-
HMAC-SHA-1 with new addition	933	7.2	-

Table 3. Running time in thousands of clock-cycles and penalty factor for HMAC-SHA-1 on a 32-bit processor. The last column denotes the table size (in bytes) required for Debraize's algorithm.

round keys (equal to the number of rounds) using Key Expansion procedure. Each round operates on 128-bit data and consists of following operations: Addition modulo 2^{64}, Rotate 8 bits right, Rotate 3 bits left and XOR. More precisely, given $x[2i]$, $x[2i+1]$ as input (with each of them 64-bit long), the round i does the following:

$$x[2i+2] = (\mathsf{RotateRight}(x[2i], 8) + x[2i+1]) \oplus key[i]$$
$$x[2i+3] = (\mathsf{RotateLeft}(x[2i+1], 3)) \oplus x[2i+2]$$

Similar to HMAC-SHA-1, we applied all the algorithms to protect SPECK 128/128 against first-order attacks. If we use conversion algorithms, we need to perform two Boolean to arithmetic conversions and one arithmetic to Boolean conversion per round; hence 64 Boolean to arithmetic conversions and 32 arithmetic to Boolean conversions overall. On the other hand, when we perform addition directly on the Boolean shares, we need 32 additions in total. We summarize the performance of all the algorithms in Table 4.

	Time	Penalty Factor	Memory
SPECK unmasked	2047	1	-
SPECK with Goubin's conversion	63550	31	-
SPECK with Debraize's conversion ($\ell = 4$)	61603	30	32
SPECK with Debraize's conversion ($\ell = 8$)	37718	18	1024
SPECK with new conversion	51134	24	-
SPECK with Goubin's addition [KRJ14]	62942	30	-
SPECK with new addition	48574	23	-

Table 4. Running time in clock-cycles and penalty factor for SPECK on a 32-bit processor. The last column denotes the table size (in bytes) required for Debraize's algorithm.

As we can see our algorithms outperform Goubin and Debraize's algorithm ($\ell = 4$), but not Debraize's algorithm for $\ell = 8$, as for HMAC-SHA-1.

10 Conclusion

We have described a new conversion algorithm from arithmetic to Boolean masking with complexity $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$ for Goubin's algorithm. We have also described a variant for performing the arithmetic addition modulo 2^k directly with Boolean shares, still with complexity $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$. Our algorithms are proved secure against first-order attacks. In practice, for arithmetic additions modulo 2^{32} as in case of HMAC-SHA-1, we obtain similar performances as Goubin's algorithms and Debraize's algorithm.

References

[BN05] Yoojin Beak and Mi-Jung Noh. Differetial power attack and masking method. *Trends in Mathematics*, 8:1–15, 2005.

[BSS+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In *Cryptographic Hardware and Embedded Systems - CHES*, pages 188–205, 2014.

[CT03] Jean-Sébastien Coron and Alexei Tchulkine. A new algorithm for switching from arithmetic to boolean masking. In *CHES*, pages 89–97, 2003.

[Deb12] Blandine Debraize. Efficient and provably secure methods for switching from arithmetic to boolean masking. In *CHES*, pages 107–121, 2012.

[Gou01] Louis Goubin. A sound method for switching between boolean and arithmetic masking. In *CHES*, pages 3–15, 2001.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, pages 463–481, 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

[KRJ14] Mohamed Karroumi, Benjamin Richard, and Marc Joye. Addition with blinded operands. In *COSADE*, 2014.

[KS73] Peter M Kogge and Harold S Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *Computers, IEEE Transactions on*, 100(8):786–793, 1973.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[NIS95] NIST. Secure hash standard. In *Federal Information Processing Standard, FIPA-180-1*, 1995.

[NP04] Olaf Neiße and Jürgen Pulkus. Switching blindings with a view towards idea. In *CHES*, pages 230–239, 2004.

[NW97] Roger M. Needham and David J. Wheeler. Tea extentions. In *Technical report, Computer Laboratory, University of Cambridge*, 1997.

[PMO07] Thomas Popp, Stefan Mangard, and Elisabeth Oswald. Power analysis attacks and countermeasures. *IEEE Design and Test of Computers*, 24(6):535–543, 2007.

A Goubin's Arithmetic-to-Boolean Conversion

From Theorem 2, one obtains the following corollary.

Corollary 1 ([Gou01]) *For any random $\gamma \in \mathbb{F}_{2^k}$, if we assume $x' = (A + r) \oplus r$, we also have $x' = A \oplus 2\gamma \oplus t_{k-1}$, where t_{k-1} can be obtained from the following recursion formula:*

$$\begin{cases} t_0 = 2\gamma \\ \forall i \geq 0, t_{i+1} = 2[t_i \wedge (A \oplus r) \oplus \omega] \end{cases} \tag{13}$$

where $\omega = \gamma \oplus (2\gamma) \wedge (A \oplus r) \oplus A \wedge r$.

Since the iterative computation of t_i contains only XOR and AND operations, it can easily be protected against first-order attacks. This gives the algorithm below.

Algorithm 7 Goubin A→B Conversion

```
Input: $A, r$ such that $x = A + r$
Output: $x', r$ such that $x' = x \oplus r$
$\gamma \leftarrow rand(k)$
$T \leftarrow 2\gamma$
$x' \leftarrow \gamma \oplus r$
$\Omega \leftarrow \gamma \wedge x'$
$x' \leftarrow T \oplus A$
$\gamma \leftarrow \gamma \oplus x'$
$\gamma \leftarrow \gamma \wedge r$
$\Omega \leftarrow \Omega \oplus \gamma$
$\gamma \leftarrow T \wedge A$
$\Omega \leftarrow \Omega \oplus \gamma$
for $j := 1$ to $k - 1$ do
    $\gamma \leftarrow T \wedge r$
    $\gamma \leftarrow \gamma \oplus \Omega$
    $T \leftarrow T \wedge A$
    $\gamma \leftarrow \gamma \oplus T$
    $T \leftarrow 2\gamma$
end for
$x' \leftarrow x' \oplus T$
```

We can see that the total number of operations in the above algorithm is $5k + 5$, in addition to one random number generation. Karroumi *et al.* recently improved Goubin's conversion scheme down to $5k+1$ operations [KRJ14]. More precisely they start the loop in (13) from $i = 2$ instead of $i = 1$, and compute t_1 directly with a single operation, which decreases the number of operations by 4.

Karroumi *et al.* also provided an algorithm to compute first-order secure addition on Boolean shares using Goubin's recursion formula, requiring $5k+8$ operations and a single random generation. More precisely, given two sensitive variables x and y masked as $x = x' \oplus s$ and $y = y' \oplus r$, their algorithm computes two shares $z_1 = (x + y) \oplus r \oplus s,\ z_2 = r \oplus s$ using Goubin's recursion formula (1); we refer to [KRJ14] for more details.

B Proof of Lemma 1

We consider again recursion (7):

$$\begin{cases} z_0 = b_0 \\ \forall i \geq 1,\ z_i = a_i z_{i-1} + b_i \end{cases}$$

The recursion for c_i is similar when we denote the AND operation by a multiplication, and the XOR operation by an addition:

$$\begin{cases} c_0 = 0 \\ \forall i \geq 1,\ c_i = a_{i-1} c_{i-1} + b_{i-1} \end{cases}$$

Therefore we obtain $c_{i+1} = z_i$ for all $i \geq 0$. From the $Q(m, n)$ function given in (8) we define the sequences:

$$G_{i,j} := Q\big(j, \max(j - 2^i + 1, 0)\big)$$

$$P_{i,j} := \prod_{v=\max(j-2^i+1,0)}^{j} a_v$$

We show that these sequences satisfy the same recurrence (10) from Section 3.2. From (8) we have the recurrence for $j \geq 2^{i-1}$:

$$\begin{aligned}
G_{i,j} &= \sum_{u=\max(j-2^i+1,0)}^{j} \left(\prod_{v=u+1}^{j} a_v \right) b_u \\
&= \sum_{u=\max(j-2^i+1,0)}^{j-2^{i-1}} \left(\prod_{v=u+1}^{j} a_v \right) b_u + \sum_{u=j-2^{i-1}+1}^{j} \left(\prod_{v=u+1}^{j} a_v \right) b_u \\
&= \left(\prod_{v=j-2^{i-1}+1}^{j} a_v \right) \sum_{u=\max(j-2^i+1,0)}^{j-2^{i-1}} \left(\prod_{v=u+1}^{j-2^{i-1}} a_v \right) b_u + Q(j, j-2^{i-1}+1) \\
&= P_{i-1,j} \cdot Q\big(j-2^{i-1}, \max(j-2^i+1,0)\big) + G_{i-1,j} \\
&= P_{i-1,j} \cdot G_{i-1,j-2^{i-1}} + G_{i-1,j}
\end{aligned}$$

We obtain a similar recurrence for $P_{i,j}$ when $j \geq 2^{i-1}$:

$$\begin{aligned}
P_{i,j} &= \prod_{v=\max(j-2^i+1,0)}^{j} a_v \\
&= \left(\prod_{v=\max(j-2^i+1,0)}^{j-2^{i-1}} a_v \right) \cdot \left(\prod_{v=j-2^{i-1}+1}^{j} a_v \right) = P_{i-1,j-2^{i-1}} \cdot P_{i-1,j}
\end{aligned}$$

In summary we obtain for $j \geq 2^{i-1}$ the relations:

$$\begin{cases} G_{i,j} = P_{i-1,j} \cdot G_{i-1,j-2^{i-1}} + G_{i-1,j} \\ P_{i,j} = P_{i-1,j} \cdot P_{i-1,j-2^{i-1}} \end{cases}$$

which are exactly the same as (10) from Section 3.2. Moreover for $0 \leq j < 2^{i-1}$, as in Section 3.2, we have $G_{i,j} = Q(j,0) = G_{i-1,j}$ and $P_{i,j} = P_{i-1,j}$. Finally we have the same initial conditions $G_{0,j} = Q(j,j) = b_j = x^{(j)} \wedge y^{(j)}$ and $P_{0,j} = a_j = x^{(j)} \oplus y^{(j)}$. This proves that the sequence $G_{i,j}$ defined by (10) in Section 3.2 is such that:

$$G_{i,j} = Q\big(j, \max(j-2^i+1,0)\big)$$

This implies that we have $G_{0,0} = Q(0,0) = z_0$ and $G_{i,j} = Q(j,0) = z_j$ for all $2^{i-1} \leq j < 2^i$. Moreover as noted initially we have $c_{j+1} = z_j$ for all $j \geq 0$. Therefore the recurrence from Section 3.2 indeed computes the carry signal c_j, with $c_0 = 0$, $c_1 = G_{0,0}$ and $c_{j+1} = G_{i,j}$ for $2^{i-1} \leq j < 2^i$. This terminates the proof of Lemma 1. □