# A New Algorithm for Switching from Arithmetic to Boolean Masking

Jean-Sébastien Coron and Alexei Tchulkine

Gemplus Card International
34 rue Guynemer, 92447 Issy-les-Moulineaux, France
{jean-sebastien.coron, alexei.tchulkine}@gemplus.com

**Abstract.** To protect a cryptographic algorithm against Differential Power Analysis, a general method consists in masking all intermediate data with a random value. When a cryptographic algorithm combines boolean operations with arithmetic operations, it is then necessary to perform conversions between boolean masking and arithmetic masking. A very efficient method was proposed by Louis Goubin in [6] to convert from boolean masking to arithmetic masking. However, the method in [6] for converting from arithmetic to boolean masking is less efficient. In some implementations, this conversion can be a bottleneck. In this paper, we propose an improved algorithm to convert from arithmetic masking to boolean masking. Our method can be applied to encryption schemes such as IDEA and RC6, and hashing algorithms such as SHA-1.

## 1 Introduction

The concept of Differential Power Analysis was introduced by Paul Kocher and al. in 1998 [7,8]. It consists in extracting information about the secret key of a cryptographic algorithm, by studying the power consumption of the electronic device during the execution of the algorithm. The attack was first described on the DES encryption scheme, then extended to other symmetrical cryptosystems such as the AES candidates [2], and also to public-key cryptosystems [5,11].

Subsequently, some countermeasures have been developed. In [3], Chari and al. proposed an approach which consists in splitting all the intermediate variables into a given number of shares, so that the power leakage of an individual share does not reveal any information to the attacker. They show that the number of power curves needed to mount an attack grows exponentially with the number of shares. A similar approach was also proposed by Goubin and al. in [5]. The drawback of this approach is that it greatly increases the computation time and the memory needed. This is a crucial issue for constrained environments such as smart-cards.

Actually, when only two shares are used, this approach consists in masking all intermediary data with a random. This technique was evaluated by Messerges in [10] for the five remaining AES candidates. For algorithms that combine boolean and arithmetic operations, two different kinds of masking must be used:

C.D. Walter et al. (Eds.): CHES 2003, LNCS 2779, pp. 89–97, 2003.

boolean masking and arithmetic masking. This is typically the case for encryption schemes such as IDEA [9] and RC6 [12], and hashing algorithms such as SHA-1 [13]. It is therefore necessary to perform conversions between boolean masking and arithmetic masking. The conversion itself must also be resistant against Differential Power Analysis. Messerges proposed in [10] an algorithm for converting between boolean masking to arithmetic masking and conversely. However, it was shown in [4] that both conversions were vulnerable to a more sophisticated Differential Power Analysis.

A new conversion algorithm was proposed by Goubin in [6]. In both directions, the conversion algorithm is such that all intermediary variable is randomly distributed; therefore, the conversion is provably resistant to first order DPA, in which no attempt is made to correlate the power consumption at different execution times. Moreover, the conversion from boolean masking to arithmetic masking is very efficient. However, the conversion from arithmetic masking to boolean masking is less efficient, as it requires a number of operations linear in the bit-size of the data to be masked. This conversion can be a bottleneck in some implementations. In this paper, we propose a secure and efficient method to convert from arithmetic masking to boolean masking.

## 2 Definitions

### 2.1 Boolean Masking and Arithmetic Masking

In this section we recall some basic definitions. We assume that the size of all intermediate variables is $k$ bits. A typical value for $k$ is 32 bits, as for SHA-1 and MD-5. The masking technique introduced in [3] consists in splitting each intermediate data that appears in the cryptographic algorithm. Then, an attacker must analyze multiple point distributions, which requires a number of power curves exponential in the number of shares. As in [10], we apply this technique with two shares. For algorithms that combine boolean and arithmetic functions, two different kinds of masking have to be used :

**Definition 1.** *We say that a data $x$ has a boolean masking when $x$ is written as $x = x' \oplus r$ where $r$ is uniformly distributed.*

For example, assume that given $x_1$, $x_2$, we must compute $x_3 = x_1 \oplus x_2$ in a secure way. Then from the masked values $x'_1$ and $x'_2$, such that $x_1 = x'_1 \oplus r_1$ and $x_2 = x'_2 \oplus r_2$, we compute the two shares $x'_3 = x'_1 \oplus x'_2$ and $r_3 = r_1 \oplus r_2$, so that $x_3 = x'_3 \oplus r_3$. Each intermediary variable is then uniformly distributed, and the procedure is resistant against first order DPA.

**Definition 2.** *We say that a data $x$ has an arithmetic masking when $x$ is written as $x = A + r \mod 2^k$ where $k$ is the size of the register and $r$ is uniformly distributed.*

For example, assume that given $x_1$, $x_2$, we must compute $x_3 = x_1 + x_2$ in a secure way. Then from the masked values $x'_1$ and $x'_2$, such that $x_1 = x'_1 + r_1$ and

$x_2 = x'_2 + r_2$, we compute the two shares $x'_3 = x'_1 + x'_2$ and $r_3 = r_1 + r_2$, so that $x_3 = x'_3 + r_3$.

For algorithms that combine boolean operations and arithmetic operations, it is therefore necessary to provide a secure conversion algorithms in both directions.

### 2.2 From Boolean Masking to Arithmetic Masking

A very efficient method for converting from boolean masking to arithmetic masking is given in [6]. It requires a number of elementary operations which is independent from the $k$, the data bit-size. The method is based on the fact that for all $x'$ the function

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

is affine in $r$, which means that for all $x', r_1, r_2$,

$$f_{x'}(r_1 \oplus r_2) = f_{x'}(r_1) \oplus f_{x'}(r_2) \oplus x'$$

Therefore, given $x', r$ such that $x = x' \oplus r$, we generate a random $k$-bit integer $r_1$, and we can compute $A = x - r \mod 2^k$ as:

$$A = f_{x'}(r) = f_{x'}((r_1 \oplus r) \oplus r_1) = f_{x'}(r_1 \oplus r) \oplus (f_{x'}(r_1) \oplus x')$$

Since $r_1$ and $r_1 \oplus r$ have the uniform distribution, the conversion is resistant against DPA. We refer to [6] for the proof that $f$ is affine and for a detailed description of the algorithm.

### 2.3 From Arithmetic to Boolean Masking

Louis Goubin proposed in [6] a method for converting from arithmetic to boolean masking, but the method is less efficient than from boolean to arithmetic. In particular, it requires a number of operations linear in the size of the registers; namely for a $k$-bit register, the number of $k$-bit operations is $5k + 5$.

## 3 Our Conversion Algorithm

We propose a new conversion algorithm from arithmetic to boolean masking which is generally more efficient than Goubin's method. Our method is based on pre-computed tables. First, we describe our method for small register size $k$ (typically, $k = 4$).

### 3.1 Conversion with Small Register Size

The algorithm uses a pre-computed table $G$ of $2^k$ variables of $k$ bits.

**Algorithm 1**: table $G$ generation.
Output: a table $G$ and a random $r$.

1. Generate a random $k$-bit $r$.
2. For $A = 0$ to $2^k - 1$ do
   $G[A] \leftarrow (A + r) \oplus r$
3. Output $G$ and $r$.

Using this table, it is easy to convert from arithmetic to boolean masking:

**Algorithm 2**: conversion from arithmetic to boolean masking.
Input: $(A, r)$, such that $x = A + r$.
Output: $(x', r)$, such that $x = x' \oplus r$.

1. Return $x' = G[A]$.

It is clear that the algorithm is resistant to first-order DPA, as all intermediary variables have the uniform distribution. In the following table, we compare our algorithm with Goubin's algorithm. The pre-computation time and conversion time is measured in number of $k$-bit operations.

| | Algorithm 2 | Goubin's method |
|---|---|---|
| Pre-computation time | $2^{k+1}$ | 0 |
| Conversion time | 1 | $5k + 5$ |
| Table size | $2^k$ | 0 |

The pre-computation time and memory required is the main limitation for algorithm 2, which is only feasible for conversion with small sizes, such as for example $k = 4$ or $k = 8$ bits. However, the table has to be computed only once for each new execution of the cryptographic algorithm; any subsequent conversion will require only one operation, instead of $5k+5$ for Goubin's method. Therefore, algorithm 2 will be more efficient when the number $n$ of conversion during the execution of a cryptographic algorithm is greater than:

$$n > \frac{2^{k+1}}{5k + 4}$$

In this case, our method will be faster with a factor:

$$\frac{n \cdot (5k + 5)}{2^{k+1} + n}$$

For example, with $k = 8$ bits variable size, and $n = 24$ conversions, algorithm 2 is roughly two times faster than Goubin's method.

### 3.2 Conversion for $\ell \cdot k$-Bit Variables Using two $k$-Bit Tables

In this section, we show how to extend the previous algorithm in order to perform conversions for larger sizes. We consider variables of size $\ell \cdot k$ bits, and we use 2 tables with $2^k$ variables each. For example, for 32 bit conversions, we can take $\ell = 8$ and $k = 4$.

The idea of the algorithm is the following. We receive as input two $\ell \cdot k$-bit variables $A$ and $R$, such that $x = A + R \mod 2^{\ell \cdot k}$. Our goal is to obtain $x'$ such that $x = x' \oplus R$, in such a way that every intermediary variable has the uniform distribution. Let split $R$ into $R_1 \| R_2$, with $R_1$ of size $(\ell - 1) \cdot k$ bits, and $R_2$ of size $k$ bits. Then, given a random $k$-bit integer $r$, we let

$$A \leftarrow (A - r) + R_2 \mod 2^{\ell k}$$

Splitting $A$ into $A_1 \| A_2$, where $A_1$ is of size $(\ell - 1) \cdot k$ bits, we now have:

$$x = (A_1 \| A_2) + (R_1 \| r) \mod 2^{\ell k}$$

Then, if $A_2 + r \geq 2^k$, we let $A_1 \leftarrow A_1 + 1 \mod 2^{(\ell-1)k}$. This is equivalent to computing the carry from the addition $A_2 + r$ and then adding this carry to $A_1$. Then, splitting $x$ into $x_1 \| x_2$, where $x_1$ is of size $(\ell - 1) \cdot k$ bits, we have:

$$x_1 = A_1 + R_1 \mod 2^{(\ell-1)k} \text{ and } x_2 = A_2 + r \mod 2^k$$

Then we can use the table $G$ generated by algorithm 2 to convert $x_2$ from arithmetic masking to boolean masking. More precisely, we let $x'_2 \leftarrow G[A_2]$, which gives:

$$x_2 = x'_2 \oplus r$$

Then we let $x'_2 \leftarrow (x'_2 \oplus R_2) \oplus r$ so that:

$$x_2 = x'_2 \oplus R_2$$

Then we apply the same method recursively to $(A_1, R_1)$ in order to obtain $x'_1$ such that $x_1 = x'_1 \oplus R_1$, so that letting $x' = x'_1 \| x'_2$, we have:

$$x = x' \oplus R$$

as required.

Actually, we can not compute the carry from $A_2 + r$ directly, because this would leak some information about $x$. Instead, we use a randomized carry table $C$, computed in the following way:

**Algorithm 3**: carry table $C$ generation.
Input: a random $r$ of $k$ bits.
Output: a table $C$ and a random $\gamma$ of $k$ bits.

1. Generate a random $k$-bit $\gamma$.
2. For $A = 0$ to $2^k - 1$ do
$$C[A] \leftarrow \begin{cases} \gamma, \text{if } A + r < 2^k \\ \gamma + 1 \mod 2^k, \text{if } A + r \geq 2^k \end{cases}$$
3. Output $C$ and $\gamma$.

Then, instead of testing if $A_2 + r \geq 2^k$, we let:

$$A_1 \leftarrow A_1 + C[A_2] - \gamma \mod 2^{(\ell-1)k}$$

This gives the following conversion algorithm, based on the pre-computed tables $G$ and $C$ of algorithms 1 and 3:

**Algorithm 4**: Conversion with $\ell \cdot k$ bit variable:
Input: $(A, R)$, such that $x = A + R$, and $r, \gamma$ generated from algorithms 1 and 3.
Output: $x'$, such that $x = x' \oplus R$.

1. $A \leftarrow A - r \mod 2^{\ell k}$.
2. Let denote $R = R_1 \| R_2$, where $R_1$ is of size $(\ell - 1)k$ bits.
3. Let $A \leftarrow A + R_2 \mod 2^{\ell k}$
4. If $\ell = 1$, then let $x' \leftarrow G[A] \oplus R_2$, then $x' \leftarrow x' \oplus r$ and return $x'$.
5. Otherwise, let $A = A_1 \| A_2$
6. Let $A_1 \leftarrow A_1 + C[A_2] \mod 2^{(\ell-1)k}$
7. Let $A_1 \leftarrow A_1 - \gamma \mod 2^{(\ell-1)k}$
8. Let $x'_2 \leftarrow G[A_2] \oplus R_2$.
9. Let $x'_2 \leftarrow x'_2 \oplus r$.
10. Apply algorithm 4 recursively with $(A_1, R_1)$ to obtain $x'_1$.
11. Return $x' = x'_1 \| x'_2$

As previously, this conversion method is resistant to first-order DPA, because all intermediary variables have the uniform distribution. We want to compare the efficiency of our method with Goubin's method. The drawback of our method is that we need to pre-compute two tables of $2^k$ values. The advantage of our method is that some computation is done on small $k$-bits variables, whereas Goubin's method always works with full $\ell \cdot k$ bits variables. Therefore, we must take into account the register size of the micro-processor. Our method is likely to be more advantageous on a 8-bit microprocessor, which is now the most common smart-card platform, than on a 32-bit microprocessor.

To make a practical comparison, we take $k = 4$, and we distinguish two kinds of microprocessor: 8-bit and 32-bit, and two variable sizes: 8-bits and 32-bits. We take $k = 4$ because the method is easier to implement for for this value of $k$, but a better trade-off may be possible. We assume that an elementary operation on a 32 bit variables requires 4 elementary operations on a 8 bit microprocessor. For example, Goubin's method on 32-bit variables on a 8 bit microprocessor will require $4 \cdot (5 \cdot 32 + 5) = 660$ operations. More generally, we denote by $T_{i,j}$ (resp. $G_{i,j}$) our method (resp. Goubin's method) for $i$-bit variables with a $j$-bit microprocessor. The following table summarizes the number of steps in all possible cases:

| | $T_{8,8}$ | $T_{8,32}$ | $T_{32,8}$ | $T_{32,32}$ | $G_{8,8}$ | $G_{8,32}$ | $G_{32,8}$ | $G_{32,32}$ |
|---|---|---|---|---|---|---|---|---|
| Pre-computation time | 64 | 64 | 64 | 64 | 0 | 0 | 0 | 0 |
| Conversion time | 10 | 10 | 76 | 40 | 45 | 45 | 660 | 165 |
| Table size | 32 | 32 | 32 | 32 | 0 | 0 | 0 | 0 |

As previously, the efficiency improvement depends on how frequently we re-compute the randomized tables. If we compute the randomized tables only once

at the beginning of the cryptographic algorithm, then our method will always be more efficient if there are at least two subsequent conversions. But if we choose to re-compute the tables before each conversion, then Goubin's method is more efficient for 8-bit variables, whereas our method is more efficient for 32-bit variables. Our method is particularly advantageous for 32-bit conversions on a 8-bit microprocessor: our method ($64 + 76$ operations) is then 4.7 times faster than Goubin's method (660 operations).

# 4 Application to SHA-1

## 4.1 Overview of SHA-1

SHA-1 is a hash function introduced by the American National Institute for Standards and Technology [13] in 1995. The description of SHA-1 consists of a general iteration procedure based on a compression function

$$F : \{0,1\}^{512} \times \{0,1\}^{160} \to \{0,1\}^{160}$$

In the following we give a very general overview of the algorithm (see [13] for details).

**General Iteration Procedure:**

1. Pad the message, so that its length is a multiple of the size of the compression function, that is 512 bits.
2. Initialize the five 32-bit chaining variables $A, B, C, D, E$ with a given $IV$ value.
3. For each message block $M$ of 512 bits, let
$$(A, B, C, D, E) \leftarrow F(M, (A, B, C, D, E)) + (A, B, C, D, E)$$
where $F$ is the compression function.
4. Output the hash value $A\|B\|C\|D\|E$.

**Compression Function $F$:**

1. Expand the 512-bit message block $M$ into 80 words $M_i$ of 32 bits.
2. For $i = 0$ to 79 do:
$$(A, B, C, D, E) \leftarrow (M_i + \mathrm{rot}_5(A) + f_i(B, C, D) + E + K_i, \\ A, \mathrm{rot}_{30}(B), C, D)$$

where $\mathrm{rot}_j$ denotes left rotation by $j$ bits, $K_i$ are constants and:

$$\begin{array}{ll} f_i(X,Y,Z) = (X\&Y)|(\neg X\&Z), & 0 \le i \le 19 \\ f_i(X,Y,Z) = X \oplus Y \oplus Z, & 20 \le i \le 39, 60 \le i \le 79 \\ f_i(X,Y,Z) = (X\&Y)|(X\&Z)|(Y\&Z), & 40 \le i \le 59 \end{array}$$

We see that SHA-1 combines boolean operations with arithmetic operations.

### 4.2 Motivation

The SHA-1 hash function can be used for MAC algorithms, for example:

$$\mathrm{MAC}_K(x) = \text{SHA-1}(K_1 \| x \| K_2)$$

or for the HMAC [1] nested construction:

$$\mathrm{HMAC}_K(x) = \text{SHA-1}(K_2 \| \text{SHA-1}(x \| K_1))$$

where $K = K_1 \| K_2$ is a secret-key. In this case, the implementation of SHA-1 has to be made resistant against DPA, otherwise a straightforward DPA attack would recover the secret-key $K$.

### 4.3 Implementation Result

In the following, we estimate the number of elementary operations which are required to have an implementation of SHA-1 resistant against DPA. Without DPA countermeasure, each of the 80 steps in the compression function requires roughly 15 elementary 32-bit operations. The DPA countermeasure requires to split each variable into 2 shares; this leads to 30 elementary operations. Moreover, assuming that $A, B, C, D$ and $E$ have initially a boolean masking, we need to convert $f_i(B, C, D)$, $\mathrm{rot}_5(A)$ and $E$ into arithmetic masking, then the sum $M_i + \mathrm{rot}_5(A) + f_i(B, C, D) + E + K_i$ back to boolean masking. This gives 3 boolean to arithmetic conversions, each requiring 7 operations using [6], and one arithmetic to boolean conversion. Therefore, each step requires 51 elementary operations on 32-bit variables (or 204 operations on 8-bit variables)[1], together with one arithmetic to boolean conversion.

In the following table, we compare the efficiency of an implementation of SHA-1 resistant against DPA, using our arithmetic to boolean conversion method, and using Goubin's method, for 8-bit and 32-bit micro-processor. The time is measured in number of elementary operations for each of the 80 steps of the compression function. For our arithmetic to boolean conversion, we recompute the randomized tables before each new conversion. This means that using our method, a 32-bit arithmetic to boolean conversion takes 140 elementary operations on a 8-bit microprocessor, and 104 operations on a 32-bit microprocessor.

| | 8-bit micro | 32-bit micro |
|---|---|---|
| Our method | 344 | 155 |
| Goubin's method | 864 | 216 |

[1] As previously, we assume that a 32-bit operation on a 8-bit micro-processor requires 4 elementary operations.

## 5 Conclusion

We have described a new conversion algorithm from arithmetic to boolean masking, which is generally more efficient than Goubin's algorithm. Our new algorithm is particularly interesting for 32-bit conversions on a 8-bit microprocessor. For example, for SHA-1 hash function, the previous table shows that an implementation secure against DPA will be roughly 2.7 times faster using our method than using Goubin's method.

## References


1. M. Bellare, R. Canetti, and H. Krawczyk, *Keying hash functions for message authentication*, Advances in Cryptology - Crypto 96 Proceedings, Lecture Notes in Computer Science Vol. 1109, N. Koblitz ed, Springer-Verlag, 1996.
2. Suresh Chari, Charantjit S. Jutla, Josyula R. Rao and Pankaj Rohatgi, *A Cautionary Note Regarding Evaluation of AES Candidates on Smart-Cards*, in Proceedings of the Second Advanced Encryption Standard (AES) Candidate Conference, http://csrc.nist.gov/encryption/aes/round1/Conf2/aes2conf.htm, March 1999.
3. Suresh Chari, Charantjit S. Jutla, Josyula R. Rao and Pankaj Rohatgi, *Towards Sound Approaches to Counteract Power-Analysis Attacks*, in Proceedings of Advances in Cryptology – CRYPTO'99, Springer-Verlag, 1999, pp. 398–412.
4. Jean-Sebastien Coron and Louis Goubin, *On Boolean and Arithmetic Masking against Differential Power Analysis*, Proceedings of CHES 2000, LNCS 1965, pp. 231–237, Springer.
5. Louis Goubin and Jacques Patarin, *DES and Differential Power Analysis – The Duplication Method*, in Proceedings of CHES 99, Springer-Verlag, August 1999, pp. 158–172.
6. Louis Goubin, *A Sound Method for Switching between Boolean and Arithmetic Masking*, proceedings of CHES 2001, LNCS 2162, pp. 3–15, Springer.
7. Paul Kocher, Joshua Jaffe and Benjamin Jun, *Introduction to Differential Power Analysis and Related Attacks*, available at `www.cryptography.com/dpa/technical`, 1998.
8. Paul Kocher, Joshua Jaffe and Benjamin Jun, *Differential Power Analysis*, in Proceedings of Advances in Cryptology – CRYPTO'99, Springer-Verlag, 1999, pp. 388–397.
9. X. Lai and J. Massey, *A Proposal for a New Block Encryption Standard*, in Advances in Cryptology – EUROCRYPT '90 Proceedings, Springer-Verlag, 1991, pp. 389–404.
10. Thomas S. Messerges, *Securing the AES Finalists Against Power Analysis Attacks*, in Proceedings of FSE 2000, Springer-Verlag, April 2000.
11. Thomas S. Messerges, Ezzy A. Dabbish and Robert H. Sloan, "Power Analysis Attacks of Modular Exponentiation in Smartcards", in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, Springer-Verlag, August 1999, pp. 144–157.
12. R.L. Rivest, M.J.B. Robshaw, R. Sidney and Y.L. Yin, *The RC6 Block Cipher*, v1.1, August 20, 1998.
13. FIPS PUB 180-1, Secure Hash Standard, *U.S. department of commerce/National Institute of Standards and Technology*