# Side-channel Analysis of Six SHA-3 Candidates[*]

Olivier Benoît and Thomas Peyrin

Ingenico, France
{forename.name}@ingenico.com

**Abstract.** In this paper we study six 2nd round `SHA-3` candidates from a side-channel cryptanalysis point of view. For each of them, we give the exact procedure and appropriate choice of selection functions to perform the attack. Depending on their inherent structure and the internal primitives used (Sbox, addition or XOR), some schemes are more prone to side channel analysis than others, as shown by our simulations.

**Key words:** side-channel, hash function, cryptanalysis, HMAC, SHA-3.

## 1   Introduction

Hash functions are one of the most important and useful tools in cryptography. A $n$-bit cryptographic hash function $H$ is a function taking an arbitrarily long message as input and outputting a fixed-length hash value of size $n$ bits. Those primitives are used in many applications such as digital signatures or key generation. In practice, hash functions are also very useful for building Message Authentication Codes (MAC), especially in a HMAC [5, 34] construction. HMAC offers a good efficiency considering that hash functions are among the fastest bricks in cryptography, while its security can be proven if the underlying function is secure as well [4].

In recent years, we saw the apparition of devastating attacks [39, 38] that broke many standardized hash functions [37, 31]. The NIST launched the `SHA-3` competition [33] in response to these attacks and in order to maintain an appropriate security margin considering the increase of the computation power or further cryptanalysis improvements. The outcome of this competition will be a new hash function security standard to be determined in 2012 and 14 candidates have been selected to enter the 2nd round of the competition (among 64 submissions).

Differential and Simple Power Analysis (DPA and SPA) were introduced in 1998 by Kocher *et al.* [26] and led to a powerful class of attacks called side-channel analysis. They consist in two main steps. First, the power consumption, the electro-magnetic signal [1] or any others relevant physical leakage from an integrated circuit is measured during multiples execution of a cryptographic algorithm. Then, one performs a hypothesis test on subkeys given the algorithm specification, the algorithm input and/or output values and the traces obtained during the first step. This second step requires to compute an intermediary results of the algorithm for a given key guess and all the input/output and analyze correlation [13] with the actual experimental traces. The intermediary result is the output of what we will call hereafter the "selection function".

Because of the widely developed utilization of HMAC (or any MAC built upon a hash function) in security applications, it makes sense to consider physical security of hash functions [28, 17, 29, 19, 35, 20]. Indeed, those functions usually manipulate no secret and have been at little bit left apart from side-channel analysis for the moment. In practice, the ability to retrieve

---

[*] This is the extended version of the article published at CHES 2010.

the secret key that generates the MACs with physical attacks is a real threat that needs to be studied and such a criteria is taken in account by the NIST for the candidates selections [24].

**Our contributions.** We present a side-channel analysis of six hash functions selected to the 2nd round of the `SHA-3` competition: `ECHO` [7], `Grøstl` [18], `SHAvite-3` [11] (three `AES`-based hash functions), `BLAKE` [3], `CubeHash` [9] and `HAMSI` [36]. This paper aims at finding the appropriate selection function for each SHA-3 candidates in a MAC setting and evaluating the relative efficiency through simulations of the corresponding attacks. Then, we draw conclusions concerning the relative complexity for protecting each candidate against first order side-channel cryptanalysis.

Of course, the intend of this paper is not to show that some particular hash functions can be broken with side-channel analysis, which should be easy in general when the implementation is not protected. However, we believe there are constructions that are naturally harder to attack or easier to implement in a secure and relatively efficient way.

# 2 How to perform side-channel attacks on hash functions

## 2.1 Message Authentication Codes with hash functions

A Message Authentication Code (MAC) is an algorithm that takes as input a arbitrary long message $M$ and a secret key $K$ and outputs a fixed-length value $V = MAC(M, K)$. One requires that it should be computationally impossible for an attacker to forge a valid MAC without knowing the secret key $K$, or to retrieve any information about $K$. This primitive allows the authentication of messages between two parties sharing the same secret key. MACs can be built upon block ciphers (i.e. CBC-MAC [6]) or hash functions in the case of HMAC [5, 34]. HMAC instantiated with the hash function $H$ is defined as follows:

$$HMAC(K, M) =$$
$$H((K \oplus opad)||H((K \oplus ipad)||M))$$
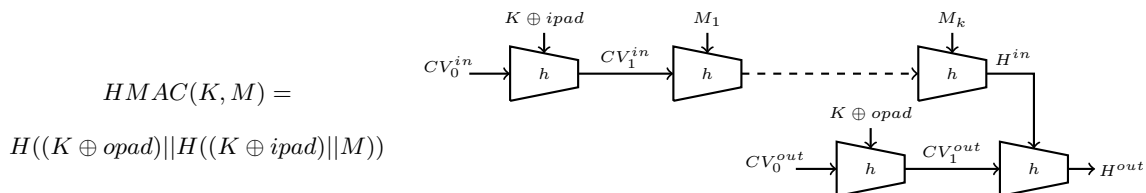


where $||$ denotes the concatenation of bit strings and $\oplus$ represents the bitwise exclusive or (XOR) boolean function. The two words $opad$ and $ipad$ are two constants of the size of a message block in the iterative hash function. This point is important: HMAC implicitly considers that $H$ is an iterative hash function. Thus, for each iteration $i$ we have an internal state (so-called chaining variable) $CV_i$ that is updated thanks to a compression function $h$ and a message block $M_i : CV_{i+1} = h(CV_i, M_i)$. The first chaining variable $CV_0$ is initialized with an initial vector $IV$ and the hash output is the very last chaining variable or a truncated version of it.

It is easy to see that when computing the first hash function call of HMAC ($H^{in} = H((K \oplus ipad)||M))$), the first iteration is $CV_1 = h(IV, K \oplus ipad)$ and one only needs to guess $CV_1$ to complete the rest of this hash function computation, whatever the message $M$. Then, for the second hash function call ($H^{out} = H((K \oplus opad)||H^{in})$), the same remark applies: one only needs to guess $CV_1$ to complete the MAC computation whatever $H^{in}$. We denote by $CV_i^{in}$ the chaining variables for the first hash call and $CV_i^{out}$ the chaining variables for the second one.

In practice, one can speed up the implementation by precomputing the $CV_1^{in}$ and $CV_1^{out}$ and starting the two hash processes with those two new initial chaining variables.

Therefore, when attacking HMAC with a side-channel technique, it is very interesting to recover $CV_1^{in}$ and $CV_1^{out}$. We are now left with the problem of being able to retrieve a fixed unknown chaining variable with random message instances. This will have to be done two times, first for $CV_1^{in}$ and then for $CV_1^{out}$. The attack process will be identical for each call, so in this article we only describe how to recover the unknown chaining variable with several known random message instances. However, note that attacking $CV_1^{in}$ should be easier than attacking $CV_1^{out}$ because in the former the attacker has full control over the message $M$, which is not the case in the latter (the incoming message for the second hash call is $H^{in}$, over which the attacker has no control). For some SHA-3 candidates, the ability to control the incoming message may reduce the number of power traces needed to recover $CV_1^{in}$. However, the maximal total improvement factor is at most 2 since the leading complexity phase remains the recovering of $CV_1^{out}$.

In the case of the so-called stream-based hash functions (for example Grindahl [25] or RadioGatún [10]), for which the message block size is rather small (smaller than the MAC key and hash output lengths), the HMAC construction is far less attractive and one uses in general the prefix-MAC construction: $MAC(K, M) = H(K||M)$. In order to avoid trivial length-extension attacks (which makes classical Merkle-Damgård [30, 14] based hash functions unsuitable for the prefix-MAC construction), the stream-based hash functions are usually composed of a big internal state (much bigger than the hash output length) and define an output function composed of blank rounds (iterations without message blocks incorporation) and a final truncation phase. However, the corresponding side-channel attack for breaking the prefix-MAC construction will not change here and our goal remains to recover the full internal state.

In this paper, we will study the 256-bit versions of the hash functions considered, but in most of the case the analysis is identical for the 512-bit versions. Moreover, when available, the salt input is considered as null. For all candidates reviewed, the message to hash is first padded and since we only give a short description of the schemes, we refer to the corresponding specification documents for all the details. Finally, since our goal is to recover the internal state before the message digesting phase, there is no need to consider potential output functions performed after all the message words have been processed.

## 2.2 Correlation side-channel attacks

Regardless of the compression function $h(CV, M)$ considered, at some point in the algorithm (usually in the very first stage), the incoming chaining variable $CV$ will be combined with the incoming message block $M$. The selection functions will be of the form:

$$w = f(cv, m)$$

where $cv$ is a subset of $CV$ and $m$ is a subset from $M$. Usually $w$, $cv$ and $m$ have the same size (8 bits in the case of AES), but strongly compressing bricks (such as the DES Sbox) may impose a smaller $w$. Ideally the selection function must be non-linear and a modification of 1-bit in one of the input should potentially lead to multiple-bit modifications in the output. For example, the output of a substitution table (Sbox) is a good selection function: block cipher encryption algorithms such as DES [16] or AES [15] are very sensitive to side-channel analysis because they both use an Sbox in their construction (a $6 \mapsto 4$-bit Sbox for DES and a $8 \mapsto 8$-bit Sbox for AES).

Some algorithms or `SHA-3` candidates (i.e. `BLAKE` or `CubeHash`) do not use such substitution table, while they rely exclusively on modular addition $\boxplus$, rotation $\lll$ and XOR $\oplus$ operations (so-called ARX constructions). In this case, side-channel analysis is still possible but the XOR or modular addition selection functions are less efficient than for the Sbox case. Moreover, it has been theoretically proven that the XOR selection function is less efficient that the modular addition operations [28]. Indeed, the propagation of the carry in the modular addition leads to some non-linearity whereas the XOR operation if completely linear. More precisely, we can quantify the efficiency difference between the `AES` Sbox, the `HAMSI` Sbox, the XOR and the modular addition selection functions by looking at the theoretical correlation results in the so-called hamming-weight model. The rest of this paper exclusively deals with the Hamming weight model since in practice this model leads to good results for the majority of the target devices.

In order to estimate the efficiency of a selection function $f(k, m)$, it is interesting to look at the theoretical correlation $c(j, r)$ between the data set $x_i$ for a key guess $j$ and the data set $y_i$ for an arbitrary real key $r$. Where $x_i = HW(f(j, m_i))$ and $y_i = HW(f(r, m_i))$, with $i \in [0, \ldots, 2^N - 1]$, $N$ being the number of bits of the selection function input message $m$ and $HW(w)$ being the Hamming Weight of the word $w$. We also denote by $\bar{x}$ (respectively $\bar{y}$) the average value of the data set $x_i$ (respectively $y_i$).

$$c(j, r) = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2} . \sqrt{\sum (y_i - \bar{y})^2}}$$

Of course, when the key guess is equal to the real key ($j = r$), we have $c(j, r) = 1$.

The Figure 1 displays $c(j, 8)$ for $j \in [0, \ldots, 255]$ for the `AES` selection function $Sbox(k, m)$, for the XOR selection function $k \oplus m$ and for the modular addition selection function $k \boxplus m$. `HAMSI` is specific because the selection function is using a subset of the Sbox for a given key, therefore, the following table displays $c(j, r)$ for $j \in [0, \ldots, 3]$ and $r \in [0, \ldots, 3]$. Only two bits of the message and two bits of the key are handled in this selection function.

| key guess / key value | $j = 0$ | $j = 1$ | $j = 2$ | $j = 3$ |
|---|---|---|---|---|
| $r = 0$ | +1.00 | −0.17 | −0.56 | −0.87 |
| $r = 1$ | −0.17 | +1.00 | +0.87 | −0.09 |
| $r = 2$ | −0.56 | +0.87 | +1.00 | +0.17 |
| $r = 3$ | −0.87 | −0.09 | +0.17 | +1.00 |

The efficiency $E(f)$ of the selection function $f$ is directly linked with the correlation contrast $c_c$ between the correct key guess (correlation = 1) and the strongest wrong key guess (correlation $= c_w$). The higher this contrast, the more efficient the selection function will be to perform a side-channel analysis. Indeed, it will be able to sustain a much higher noise level.

---

[1] In practice, if the attacker managed to characterize the chip leakage, he eventually can distinguish the wrong guess from the correct guess by taking in consideration the correlation sign (positive or negative). Note that a contrast of zero does not means that the XOR selection function is not yielding any information. Indeed, the attacker have reduced the subkey space from 256 to 2 values (with correlation 1 and -1).

$$c_c = \frac{1-|c_w|}{|c_w|}$$

| selection function | AES Sbox | *modular* addition | XOR[1] | HAMSI Sbox |
|---|---|---|---|---|
| $c_w$ | 0.23 | 0.75 | $-1$ | 0.87 |
| $c_c$ | 3.34 | 0.33 | 0 | 0.15 |

The values of $c_w$ are extracted from Figure 1 by measuring the highest correlation peak (except the peak with correlation equal to 1 which corresponds to the correct guess). The result of this theoretical/simulation study is the following:

$$E(\texttt{AES}\ \text{Sbox}) > E(\text{modular addition}) > E(\texttt{HAMSI}\ \text{Sbox}) > E(\text{XOR})$$

In the rest of this article, we will search for the best selection function for each SHA-3 candidate analyzed with this conclusion in mind.
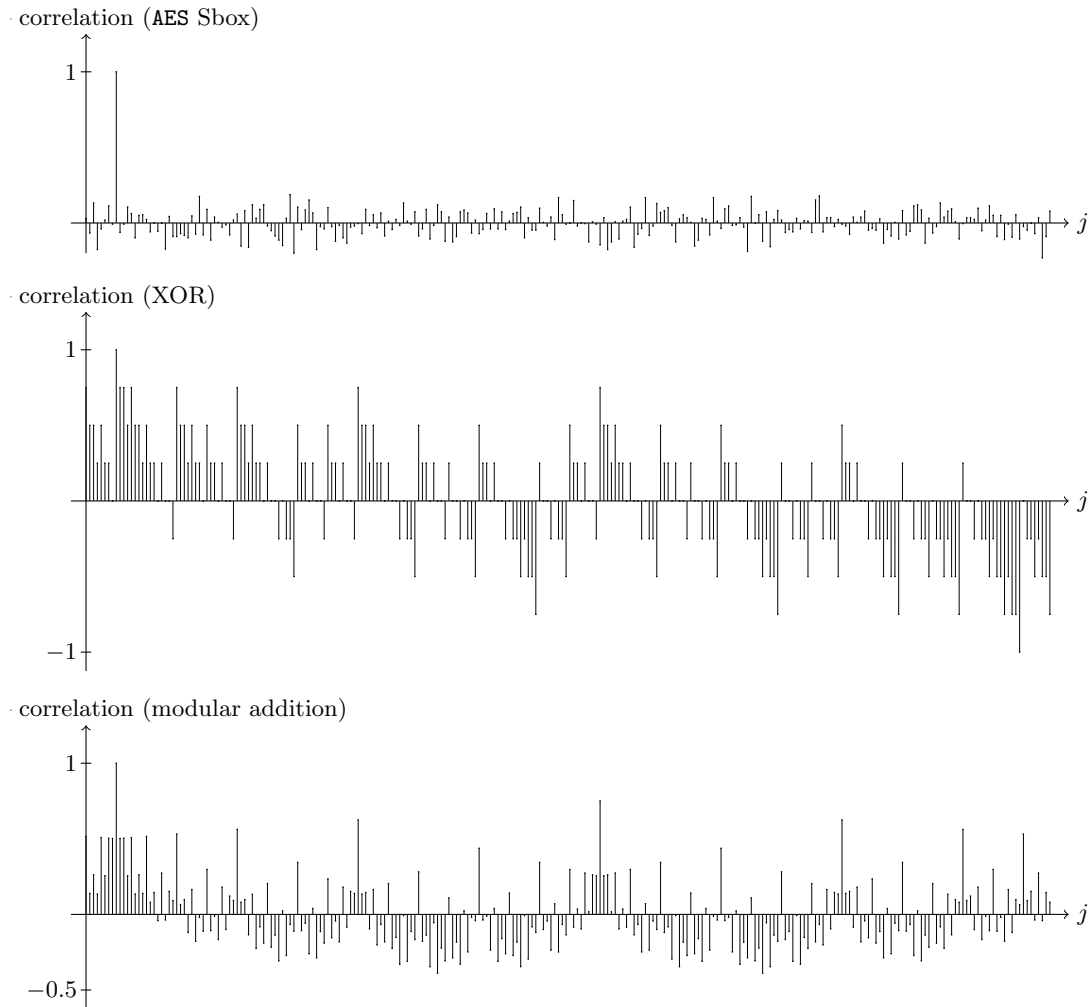


**Fig. 1.** Correlations $c(j,8)$ in the Hamming Weight model for the AES Sbox, XOR and modular addition selection function respectively.

# 3 Experimental results on elementary selection functions

## 3.1 Test protocol

In order to verify the theoretical results of the previous section regarding the relative efficiency of the four elementary selection functions that can be found in the `SHA-3` candidates, we setup a platform to experimentally perform a correlation attack on an electromagnetic side-channel. Our target is a Xilinx Spartan 3 FPGA in which we instanciate a 32-bit RISC CPU. We then execute on this CPU a software which computes a 16 bytes output $R$ from a 16 bytes input $M$ and a fixed 16 bytes key $K$. Output bytes number 0 to 3, 4 to 7, 8 to 11 and 12 to 15 are respectively the output of the following four selection function:

$$R_i = K_i \oplus M_i \qquad\qquad\qquad\qquad\qquad with\ i \in [0,3]$$
$$R_i = (K_i \boxplus M_i) mod 256 \qquad\qquad\qquad\quad with\ i \in [4,7]$$
$$R_i = Sbox_{\texttt{AES}}(K_i \oplus M_i) \qquad\qquad\qquad with\ i \in [8,11]$$
$$R_i = Sbox_{\texttt{HAMSI}}(K_i(j+1)||M_i(j+1)||K_i(j)||M_i(j))\ with\ i \in [12,15]$$

ElectroMagnetic (EM) signal is recorded during the execution of a transfert loop that reads $R_i$ and writes the value to another 16 bytes array for $i \in [0,15]$. This experiment protocol is used in order to avoid bias that could occur if we were analysing the EM signal during the actual computation of $R_i$. Indeed, these computation do not perform the same operations and therefore, depending on the position of the EM sensor, the results could be irrelevant.

For example, if the sensor is above the ALU, the two first selection functions could lead to good correlation results whereas the two selection functions based on the output of an Sbox substitution would lead to poor correlation results.

The EM signal is recorded for 100.000 executions of the above process at a sampling frequency of 5GS/s. Each input message value is repeated ten times in order to average the curves and reduce the noise. Moreover, the Lecroy oscilloscope Noise Reduction mode is used. The sensor is a small coil of approximately 1mm diameter and 10 spires (see Figure 2). A broadband amplifier with a bandwidth of 1GS/s and 30db gain is also used. Each curve weight 20kB and consists of 20.000 samples (8-bit data) . The key value has been fixed to (0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3).

## 3.2 Practical correlation analysis

With a homemade software, we perform a correlation analysis in order to retrieve the 16 bytes of the secret key. For each byte from index 0 to 11, 256 guesses are made, for each byte from index 12 to 15, only 4 guesses are made and the corresponding correlation curves are computed. We then look for the best correlation among the guesses in order to find the best key candidate. It appears that the best results are achieved by sorting our key guess according to negative correlation values rather than positive correlation values. This means that the chip leakage in EM is the complement of our model. A hamming weight of 0 leads to the highest signal level and a hamming weight of 8 leads to the lowest signal level.

On Figure 3 we can see that the correct key bytes values are found for the `AES` Sbox and the modular addition selection functions. Moreover, the contrast (last column) is in line with the theoretical values. As predicted by the theory, the XOR and `HAMSI` Sbox selection functions are less efficient and despite the use of 100k curves, the key bytes are not fully retrieved.

The Figure 4 displays the key found by correlation for a given number of curves, starting from 50 curves and ending with 10k curves. We notice that the four key bytes handled by the
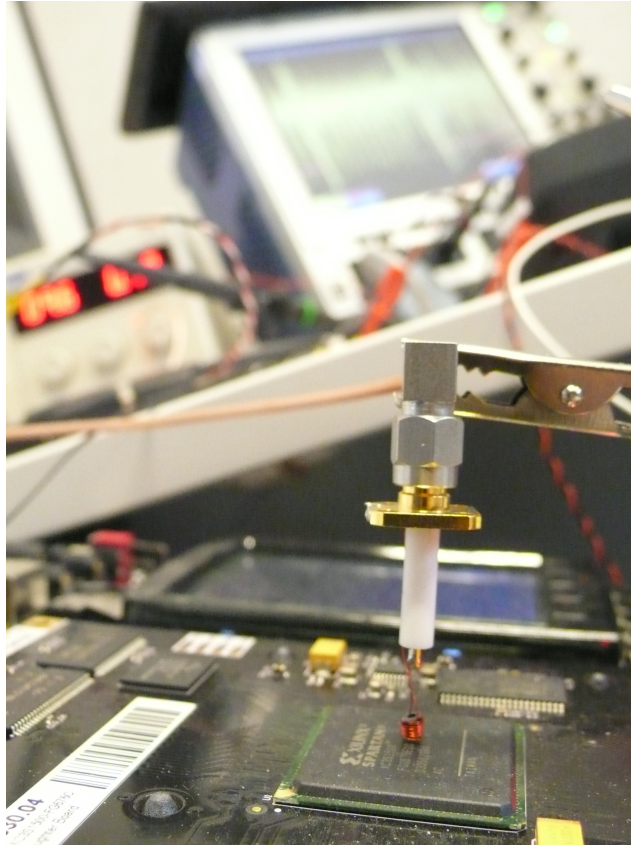
**Fig. 2.** ElectroMagnetic sensor above the target FPGA running the elementary selection functions.

```
Correlation:    XOR,    ADD,    AES    HAMSI      selection function
                0..3    4..7    8..11  12..15
Best guess selection criteria : Minimum
Previous state range: 0..0
Subkeys range:        0..255
Sample per file:      20000
Sample range:         4150..15349
Working on file ID:   0..9999
Split step: 700
Split slot size: 100
Memory requirement: 213 Mo  ,press y for cache memory y

Index  Rank 1 [x,cor]      Rank 2 [x,cor]      Rank 3 [x,cor]      Rank 4 [x,cor]      Rank 5 [x,cor]      Contrast

S00 :  00 [04209,-0.390]  02 [04209,-0.382]  08 [04209,-0.378]  0A [04209,-0.371]  10 [04210,-0.304]    1.9%
S01 :  0B [04910,-0.430]  03 [04909,-0.393]  09 [04909,-0.370]  4B [04910,-0.341]  01 [04908,-0.339]    9.2%
S02 :  08 [05610,-0.412]  00 [05610,-0.373]  0A [05609,-0.371]  48 [05610,-0.337]  02 [05609,-0.334]   10.5%
S03 :  0B [06309,-0.417]  09 [06310,-0.406]  03 [06309,-0.384]  01 [06309,-0.380]  4B [06309,-0.333]    2.6%

S04 :  00 [07010,-0.409]  FE [07010,-0.322]  02 [07010,-0.292]  F8 [07010,-0.287]  08 [07010,-0.287]   26.9%
S05 :  01 [07709,-0.323]  FF [07710,-0.283]  03 [07710,-0.254]  FB [07710,-0.242]  F9 [07709,-0.237]   14.0%
S06 :  02 [08409,-0.361]  04 [08410,-0.312]  00 [08410,-0.311]  FA [08410,-0.283]  FC [08411,-0.281]   15.8%
S07 :  03 [09109,-0.422]  01 [09110,-0.313]  FB [09110,-0.300]  0B [09110,-0.294]  83 [09109,-0.275]   34.8%

S08 :  00 [09810,-0.399]  9C [09810,-0.098]  26 [09810,-0.094]  28 [09808,-0.089]  33 [09810,-0.086]  304.8%
S09 :  01 [10508,-0.307]  32 [10509,-0.076]  0B [10507,-0.073]  09 [10509,-0.072]  9D [10508,-0.064]  301.6%
S10 :  02 [11209,-0.362]  9E [11209,-0.088]  08 [11210,-0.086]  31 [11209,-0.080]  24 [11212,-0.078]  313.2%
S11 :  03 [11910,-0.425]  AD [11910,-0.090]  25 [11908,-0.086]  CA [11910,-0.080]  9F [11910,-0.079]  374.7%

S12 :  00 [12609,-0.183]  01 [12626,-0.063]  02 [12640,-0.059]  03 [12550,-0.027]                     188.5%
S13 :  02 [13307,-0.326]  01 [13308,-0.266]  03 [13324,-0.093]  00 [13345,-0.006]                      22.3%
S14 :  03 [14010,-0.130]  01 [14049,-0.049]  00 [14026,-0.047]  02 [14049,-0.041]                     163.0%
S15 :  00 [14726,-0.257]  03 [14709,-0.233]  01 [14650,-0.140]  02 [14650,-0.124]                      10.3%
```

**Fig. 3.** Best key guesses (highest negative correlation) for each key byte.

AES Sbox selection function are retrieved wih only 250 curves. The four key bytes handled by the Modular Addition selection function are retrieved with 500 curves. The 8 key bytes handled respetively by the XOR and the HAMSI Sbox selection function are not fully retrieved whatever the number of curves.

The Figure 5 displays the correlation curves for all the wrong key guesses (green) and the correct key guess (red) for key byte 0, 4, 8, 12 (respectively handled by XOR, ADD, AES Sbox and HAMSI Sbox selection functions). We notice that there are strong correlation peaks for wrong key guesses in the case of the XOR and the HAMSI Sbox selection function. For the AES Sbox selection function, there are no correlation peaks above noise level for wrong key guesses.

| Files | S00 | S01 | S02 | S03 | S04 | S05 | S06 | S07 | S08 | S09 | S10 | S11 | S12 | S13 | S14 | S15 | contrast |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00050 : | 5E | 73 | DA | 11 | 02 | 30 | 02 | 72 | F6 | 47 | 91 | 03 | 00 | 02 | 01 | 00 | 12.2 |
| 00100 : | 00 | 01 | 48 | 03 | 00 | DF | 02 | 3D | 00 | A3 | 02 | 03 | 00 | 02 | 03 | 00 | 9.5 |
| 00150 : | 00 | 03 | 08 | 0B | 02 | FB | 02 | FB | 00 | 65 | 02 | 03 | 00 | 02 | 03 | 00 | 17.6 |
| 00200 : | 00 | 03 | 08 | 0B | 00 | FB | 00 | FB | 00 | 65 | 02 | 03 | 00 | 02 | 03 | 00 | 25.5 |
| 00250 : | 00 | 03 | 08 | 0B | 00 | FF | 00 | FB | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 32.3 |
| 00300 : | 00 | 03 | 08 | 0B | 00 | FF | 00 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 53.6 |
| 00350 : | 00 | 03 | 08 | 0B | 00 | FF | 00 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 49.9 |
| 00400 : | 00 | 03 | 08 | 0B | 00 | 01 | 00 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 57.6 |
| 00450 : | 00 | 03 | 08 | 0B | 00 | 01 | 00 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 58.7 |
| 00500 : | 00 | 03 | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 56.0 |
| 00600 : | 02 | 03 | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 03 | 58.1 |
| 00700 : | 00 | 0B | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 64.4 |
| 00800 : | 08 | 0B | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 00 | 00 | 66.1 |
| 00900 : | 0A | 0B | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 00 | 00 | 62.4 |
| 01000 : | 08 | 0B | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 00 | 00 | 76.0 |
| 02000 : | 00 | 0B | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 88.4 |
| 03000 : | 00 | 0B | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 114.2 |
| 04000 : | 00 | 0B | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 120.1 |
| 05000 : | 08 | 0B | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 127.6 |
| 06000 : | 08 | 0B | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 123.9 |
| 07000 : | 00 | 0B | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 129.6 |
| 08000 : | 00 | 0B | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 126.8 |
| 09000 : | 00 | 0B | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 136.4 |
| 10000 : | 00 | 0B | 08 | 0B | 00 | 01 | 02 | 03 | 00 | 01 | 02 | 03 | 00 | 02 | 03 | 00 | 146.7 |

**Fig. 4.** Best key candidate retrieved by correlation analysis with 50 to 10000 curves.

These results are close to the theoretical results but we would have expected to retrieve the key bytes handled by the XOR and the HAMSI Sbox selection functions by increasing the number of curves. Obviously, some additional factors come into picture that are not consider in our theoretical model. There could be some quantization issues (the present measurement have been performed with an 8-bit oscilloscope).

## 4   AES-based SHA-3 candidates

In this section, we analyze ECHO [7], Grøstl [18] and SHAvite-3 [11], three AES-based SHA-3 candidates. We recall that the round function of AES is composed of four layers (we use the order AddRoundKey, SubBytes, ShiftRows and Mixcolumns) and we refer to [32] for a complete specification of this block cipher.

### 4.1   ECHO

**Description.**   ECHO [7] is an iterated hash function whose chaining variable $CV_i$ is 512-bit long. Its compression function $h$ maps $CV_i$ and a 1536-bit message block $M_i$ to a new chaining
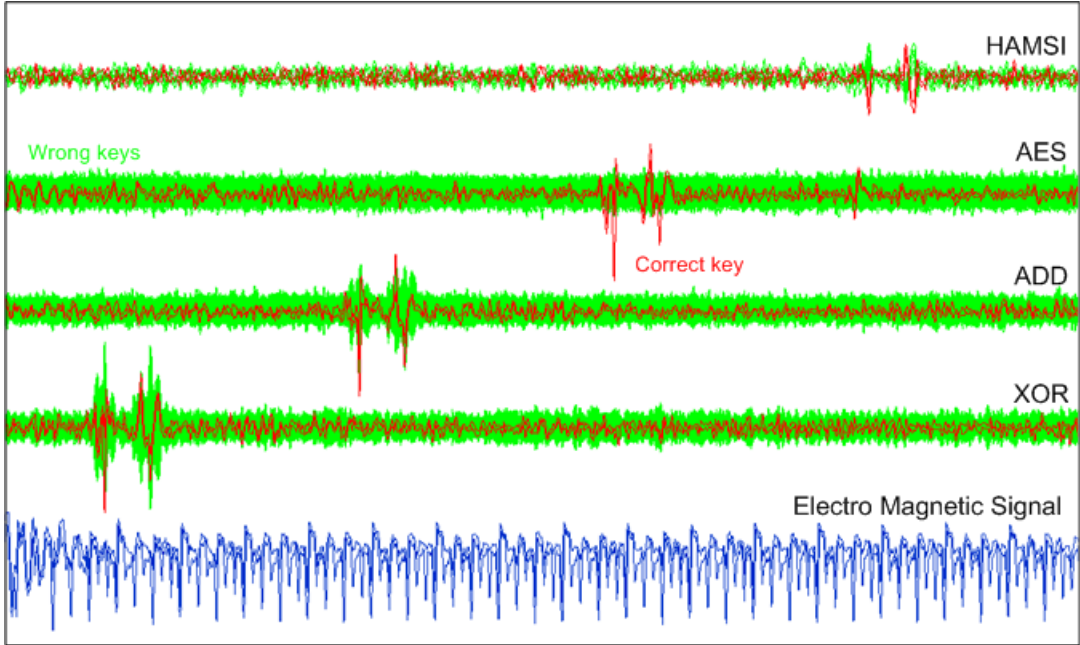
**Fig. 5.** Correlation curves for correct (red) and wrong (green) key guesses and one EMA curve of the target process (blue).

variable $CV_{i+1}$. More precisely, with $CV_i$ and $M_i$ the compression function $h$ initializes a 2048-bit internal state which is viewed as a $4 \times 4$ matrix of 128-bit words ($CV_i$ initializes the first column while $M_i$ fills the three other ones). A fixed-key permutation $P_E$ is applied to this internal state and the output chaining variable is built by calling the $\mathsf{shrink}_{256}$ function that XORs all the 512-bit columns together after a feedforward:

$$CV_{i+1} = \mathsf{shrink}_{256}(P_E(CV_i||M_i) \oplus (CV_i||M_i)).$$

The permutation $P_E$ contains 8 rounds, each composed of three functions very similar to the `AES` ones, but on 128-bit words instead of bytes. First, the BIG.SubBytes function mimics the application of 128-bit Sboxes on each state word by applying 2 `AES` rounds with fixed round keys (determined by the iteration and round numbers). Then, BIG.ShiftRows rotates the position in their matrix column of all the 128-bit words according to their row position (analog to the `AES`). Finally, BIG.MixColumns is a linear diffusion layer updating all the columns independently. More precisely, for one matrix column, it applies sixteen parallel `AES` MixColumns transformations (one for each byte position in a 128-bit word).

**Side-channel analysis.** The incoming chaining variable $CV$ fills the first 128-bit words column (denoted $cv_i$ in Figure 6) of the matrix representing the internal state, while the three other columns are filled with the known random incoming message (denoted $m_i$ in Figure 6). The goal of the attacker is therefore to retrieve the words $cv_i$.

The first layer (BIG.SubBytes) handles each 128-bit word individually. The known and secret data are not mixed yet ($cv_i \mapsto cv_i'$ and $m_i \mapsto m_i'$) and therefore it is not possible to derive a selection function at this point. The same comment applies to the second layer (BIG.ShiftRows) and one has to wait for the third layer (BIG.MixColumns) to observe known and secret data mixing: each column will depend on one secret word $cv_i'$ and three known words $m_i'$ (see Figure 6). More precisely, for each 128-bit word column, BIG.MixColumns applies

sixteen parallel and independent AES MixColumns transformations (one for each byte position) and each MixColumns call manipulates one byte of secret and three bytes of known data. Overall, in the end of the first round, every byte $w[b]$ of an internal state word $w$ (we denote $w[b]$ the $b$-th byte of $w$) can be written as the following affine equation (see the AES MixColumns definition for the $\alpha$, $\beta$, $\gamma$ and $\delta$ values):

$$w_{i_0}[b] = \alpha \cdot cv'_{i_1}[b] \oplus \beta \cdot m'_{i_2}[b] \oplus \gamma \cdot m'_{i_3}[b] \oplus \delta \cdot m'_{i_4}[b]$$

with $b \in [0, \dots, 15]$, $i_0 \in [0, \dots, 15]$, $i_1 \in [0, \dots, 3]$ and $i_2, i_3, i_4 \in [0, \dots, 11]$. One could use those $w_i[b]$ as selection functions, but the mixing operation would be the exclusive or. As already explained, the selection function involving an XOR is the least efficient one. It seems much more promising to wait the first layer from the second round of the ECHO internal permutation.
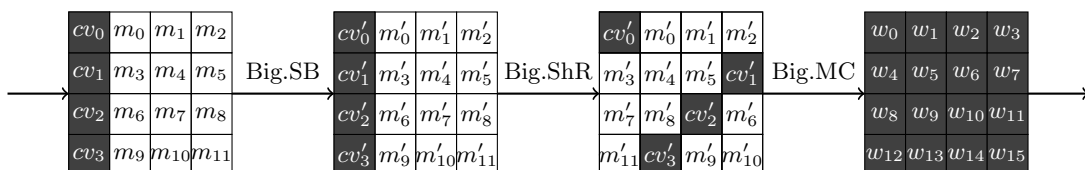


**Fig. 6.** Recovering the internal state for ECHO. The gray cells represent the words that depends on the initial secret chaining variable. Each cell represents a 128-bit word.

Indeed, the BIG.SubBytes transformation applies directly two AES rounds independently for each words $w_i$. The first function of the first AES round is the subkey incorporation and in the case of ECHO those subkeys are fully known constants (we denote them $t_i$). Then, the second function of the first AES round applies the AES Sbox to each byte of the internal state. Therefore, we obtain the words $w'_i$ on the output:

$$w'_i[b] = Sbox(w_i[b] \oplus t_i[b]).$$

These equations can be used as selection functions manipulating only the AES Sbox which is much more efficient than the XOR case. Overall, one has to perform 64 AES Sbox side-channel attacks in order to guess all the words $cv'_i$ byte per byte. By inverting the BIG.SubBytes layer from the words $cv'_i$, one recovers completely $CV$. Note that for each byte of $cv'_i$, one gets 4 selection functions involved. Thus, the overall number of curved can be reduced by a factor 4 at maximum by using and combining this extra information.

Overall, 64 correlation analysis based on the AES Sbox selection function are required.

## 4.2 Grøstl

**Description.** Grøstl [18] is an iterated hash function whose compression function $h$ maps a 512-bit chaining variable $CV_i$ and a 512-bit message block $M_i$ to a new chaining variable $CV_{i+1}$. More precisely, two fixed-key permutations $P_G$ and $Q_G$, only differing in the constant subkeys used, are applied:

$$CV_{i+1} = P_G(CV_i \oplus M_i) \oplus Q_G(M_i) \oplus CV_i.$$

Each permutation is composed of 10 rounds very similar to the AES ones, except that they update a 512-bit state, viewed as a $8 \times 8$ matrix of bytes (instead of $4 \times 4$). Namely, for each

round, constant subkeys are first XORed to the state (AddRoundConstant), then the `AES` Sbox is applied to each byte (SubBytes), the matrix rows are rotated with distinct numbers of positions (ShiftBytes) and finally a linear layer is applied to each byte column independently (MixBytes). This is depicted in Figure 7.
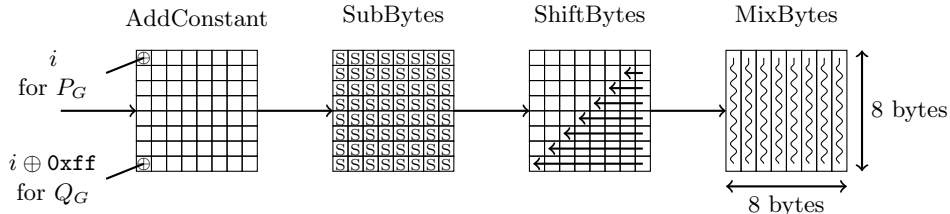


**Fig. 7.** One round of the internal permutation $P_G$ of `Grøstl`. Each cell represents a byte.

**Side-channel analysis.** The `Grøstl` case is very simple. One can see that the incoming message block $M$ is processed trough the $Q_G$ permutation. Since the output of this permutation only depends on the message block and not on the incoming chaining variable $CV$, we can completely ignore this part of the compression function. Then, the permutation $P_G$ takes as input $M \oplus CV$ and one may be tempted to perform the side-channel attack during this operation. As demonstrated earlier, it is much more convenient to wait for the opportunity to attack the `AES` Sbox instead. The first layer of the permutation $P_G$ is the AddConstant function which XORs the round number $i$ (the counting starting from 0) to the top left byte of the internal and therefore the operation is fully transparent. Then, the second layer of the first $P_G$ round is the SubBytes function which applies the `AES` Sbox to every byte of the internal state $w = M \oplus CV$:

$$w[b] = m[b] \oplus CV[b]$$

with $b \in [0, \dots, 63]$. The output state is denoted $w'$ and we obtain the following selection function which recovers $CV$ byte per byte:

$$w'[b] = Sbox(w[b]).$$

Overall, 64 correlation analysis based on the `AES` Sbox selection function are required. Note that it is possible to improve this attack when dealing with $CV_1^{in}$ (the unknown chaining variable for the first hash call) by choosing appropriately the message. More precisely, one can divide the computation time by a factor 64 when choosing all $m[b]$ as equals. Indeed, this allows to perform in parallel the side-channel analysis of the 64 unknown bytes.

### 4.3 SHAvite-3

**Description.** `SHAvite-3` [11] is an iterated hash function whose compression function $h$ maps a 256-bit chaining variable $CV_i$ and a 512-bit message block $M_i$ to a new chaining variable $CV_{i+1}$. Internally, we have a block cipher $E^S$ in classical Davies-Meyer mode

$$CV_{i+1} = CV_i \oplus E^S_{M_i}(CV_i).$$

This block cipher derives many subkeys thanks to a key schedule (all subkeys depending on the message $M_i$ only) and is composed of 12 rounds of a 2-branch Feistel scheme (128 bits per branch). The basic primitive in the Feistel rounds is the application of 3 `AES` rounds with subkeys incoming from the key schedule.

**Side-channel analysis.** For `SHAvite-3`, we divide the attack in two phases (see Figure 8). In the first one, we recover the right part (in the Feistel separation) of the incoming chaining variable ($CV^R$) during the first round. Once this first phase succeeded, we recover the left part of the incoming chaining variable ($CV^L$) during the second round. The message expansion maps the incoming message $M$ to three 128-bit message words ($m_0^j, m_1^j, m_2^j$) for each round $j$. One round $j$ of `SHAvite-3` consists in executing sequentially three `AES` round functions with as input one branch of the current `SHAvite-3` state and ($m_0^j, m_1^j, m_2^j$) as subkeys. Consequently, for the first `SHAvite-3` round, the secret vector ($CV^R$) is mixed with the known message word $m_0^1$ during the AddRoundKey layer of the first `AES` round and we note:

$$w[b] = CV^R[b] \oplus m_0^1[b].$$

with $b \in [0, \ldots, 15]$. One could use this equation as the selection function, but it is more appropriate to use the output of the very next transformation instead, i.e. the SubBytes layer:

$$w'[b] = Sbox(w[b]).$$

Before executing the second round of `SHAvite-3`, the left part of the chaining variable ($CV^L$) is XORed with the output $w''$ of the three `AES` rounds. Then, this word is mixed with $m_0^2[b]$ just before the first `AES` round of the second `SHAvite-3` round and we note:

$$z[b] = CV^L[b] \oplus w''[b] \oplus m_0^2[b].$$

Obviously, after a successful first phase, it is possible to compute $w''[b]$ and therefore $CV^L$ is the only unknown constant. Once again, one could use this equation as the selection function, but it is more appropriate to use the output of the very next transformation instead, i.e. the SubBytes layer:

$$z'[b] = Sbox(z[b]).$$

Overall, we recover byte per byte the $CV^L$ and $CV^R$ values. Only 32 correlation analysis based on the `AES` Sbox selection function are required.
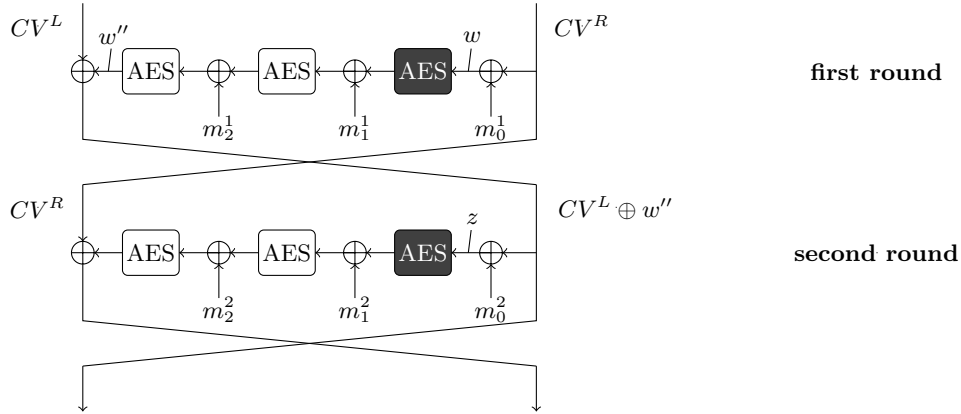


**Fig. 8.** Recovering the internal state for `SHAvite-3`. The AES rounds we use for recovering the internal state are depicted in black.

## 5  Other SHA-3 candidates

In this section, we analyze `BLAKE` [3], `CubeHash` [9] and `HAMSI` [36], three 2nd round `SHA-3` candidates.

## 5.1  BLAKE

**Description.**  BLAKE [3] is an iterated hash function whose compression function $h$ maps a 256-bit chaining variable $CV_i$ and a 512-bit message block $M_i$ to a new chaining variable $CV_{i+1}$. Internally, the update is done with a block cipher $E^B$, keyed with the message block (see Figure 9):

$$CV_{i+1} = final(E^B_{M_i}(init(CV_i)), CV_i).$$

where the *init* function initializes the 512-bit internal state with $CV_i$ and constants. The *final* function computes the output chaining variables according to $CV_i$, constants and the internal state after the application of $E^B$. The internal state is viewed as a $4 \times 4$ matrix of 32-bit words and the block cipher $E^B$ is composed of 10 rounds, each consisting of the application of eight 128-bit sub-functions $G_i$. Assume an internal state for BLAKE with $v_{i+4j}$ representing the 32-bit word located on row $j$ and column $i$, one round of $E^B$ is:

$$G_0(v_0, v_4, v_8, v_{12}) \quad G_1(v_1, v_5, v_9, v_{13}) \quad G_2(v_2, v_6, v_{10}, v_{14}) \quad G_3(v_3, v_7, v_{11}, v_{15})$$
$$G_4(v_0, v_5, v_{10}, v_{15}) \quad G_5(v_1, v_6, v_{11}, v_{12}) \quad G_6(v_2, v_7, v_8, v_{13}) \quad G_7(v_3, v_4, v_9, v_{14})$$

A sub-function $G_i$ incorporates 32-bit message chunks $m_i$ and is itself made of additions, XORs and rotations. During the round $r$, the function $G_s(a, b, c, d)$ processes the following steps:

$$a \leftarrow (a \boxplus b) \boxplus (m_i \oplus k_j)$$
$$d \leftarrow (d \oplus a) \ggg 16$$
$$c \leftarrow (c \boxplus d)$$
$$d \leftarrow (b \oplus c) \ggg 12$$
$$a \leftarrow (a \boxplus b) \boxplus (m_j \oplus k_i)$$
$$d \leftarrow (d \oplus a) \ggg 8$$
$$c \leftarrow (c \boxplus d)$$
$$d \leftarrow (b \oplus c) \ggg 7$$

where $\ggg x$ denotes the right rotation of $x$ positions, $i = \sigma_r(2s)$ and $j = \sigma_r(2s = 1)$. The notation $\sigma_r$ represents a family of permutations on $\{0, \ldots, 15\}$ defined in the specifications document.
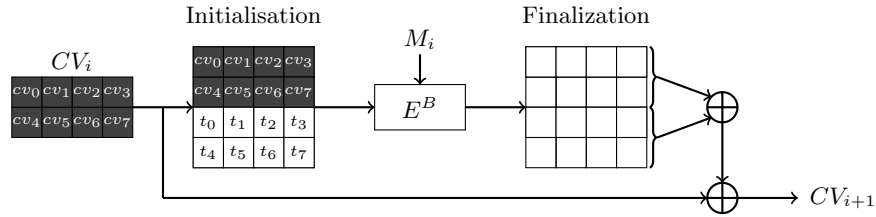


**Fig. 9.** The compression function of BLAKE.

**Side-channel analysis.**  The sixteen 32-bit internal state words are initialized with the eight secret chaining value $CV$ words (denoted $cv_i$) and constants values $t_i$ (see Figure 9). Then, the eight $G_i$ functions during the first BLAKE round are applied to the internal state and one can check that the two first parameters of $G_0$, $G_1$, $G_2$ and $G_3$ are $(cv_0, cv_1)$, $(cv_2, cv_3)$, $(cv_4, cv_5)$ and

$(cv_6, cv_7)$ respectively. Our goal is therefore to recover $a_0$ and $b_0$ when applying $G_i(a_0, b_0, c_0, d_0)$ with $0 \le i \le 3$. The functions $G_i$ consist in a sequence of eight transformations, the five first being:

$$a_1 = (a_0 \boxplus b_0) \boxplus m_k$$
$$d_1 = (d_0 \oplus a_1) \ggg 16$$
$$c_1 = c_0 \boxplus d_1$$
$$b_1 = (b_0 \oplus c_1) \ggg 12$$
$$a_2 = a_1 \boxplus b_1 \boxplus m_l$$

In practice, the first transformation will be computed in one of the three following way:

$$\text{first } a \leftarrow a \boxplus b \text{ then } a \leftarrow a \boxplus m_i$$
$$\text{first } a \leftarrow a \boxplus m_i \text{ then } a \leftarrow a \boxplus b$$
$$\text{first } x \leftarrow b \boxplus m_i \text{ then } a \leftarrow a \boxplus x$$

For the second and third case, $a_0$ and $b_0$ can be found by two side-channel analysis applied successively to the two modular addition selection function (working byte per byte):

$$w_i = cv_i \boxplus m_k \text{ and } z_i = cv_{i+1} \boxplus w_i.$$

For the first case, $a_0 \boxplus b_0$ can be recovered by performing the side-channel analysis on the second modular addition selection function. In order to solve the problem and estimate $a_0$ and $b_0$ the attacker has to target the output of the fourth transformation of $G_i$. However, in this case the selection function would be based on the XOR operation. Therefore it seems more interesting to aim for the fifth transformation of $G_i$, a modular addition.

Overall, the full chaining variable can be retrieved by exploiting exclusively the modular addition selection function. A total of 32 modular addition correlation analysis at byte level must be performed.
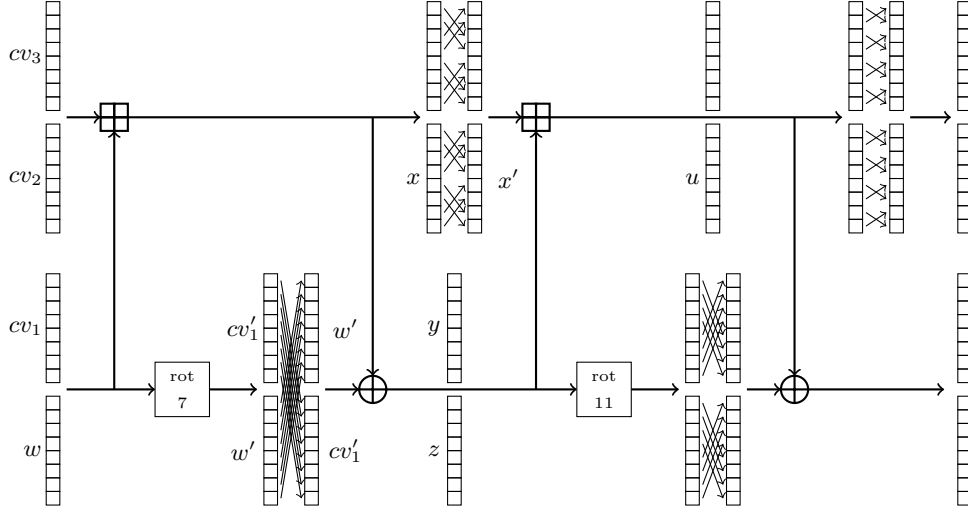
### 5.2  CubeHash

**Description.**  CubeHash [9] is an iterated hash function whose compression function $h$ maps a 1024-bit chaining variable $CV_i$ and a 256-bit message block $M_i$ to a new chaining variable $CV_{i+1}$. Internally, the update is done with a permutation $P_C$:

$$CV_{i+1} = P_C(CV_i \oplus (M_i || \{0\}^{768})).$$

The internal state is viewed as a table of 32 words of 32 bits each. The permutation $P_C$ is composed of 16 identical rounds and one round is made of ten layers (see Figure 10): two intra-word rotation layers, two XOR layers, two $2^{32}$ modular addition layers and four word positions permuting layers.

**Side-channel analysis.**  The attack is divided into 4 steps. The incoming chaining variable $CV$ fills an internal state represented by a vector of four 256-bit words or eight 32-bit words (denoted $cv_i$). The known random incoming message $M$ is first XORed with $cv_0$ and then starts the first round of permutation $P_C$. Thus, without any other alternative, the first selection function is of XOR type and recovers $cv_0$ byte per byte:

$$w[b] = cv_0[b] \oplus M[b].$$

**Fig. 10.** Recovering the internal state of `CubeHash` during one round of the internal permutation $P_C$. Each cell represents a 32-bit word.

with $b \in [0, \ldots, 7]$. Once this first step successfully performed, the attacker fully knows $w$ for a given incomming message and the first layer of $P_C$ adds each 32-bit words from $w$ to those from $cv_2$ modulo $2^{32}$ (and each words from $cv_1$ to those from $cv_3$). The second selection function is therefore of modular addition type and recovers $cv_2$ byte per byte (starting from the LSB of the modular addition):

$$x[b] = cv_2[b] \boxplus w[b].$$

The second layer of $P_C$ applies a rotation to each 32-bit word of $w$ and $cv_1$ (we denote the rotated variants $w'$ and $cv_1'$ respectively) and then XORs $x$ to this rotated version of $cv_1$:

$$z[b] = cv_1'[b] \oplus x[b]$$

This could be used as a selection function in order to retrieve $cv_1$, but being of XOR type its seems more advantageous to wait one more computation step and hope for a modular addition selection function. This can be achieved during the very next modular addition and we recover $cv_1'$ byte per byte:

$$u[b] = (cv_1'[b] \oplus x[b]) \boxplus x'[b].$$

Note that one may think that this selection function is of XOR type, but during the attack the bytes $x[b]$ and $x'[b]$ are independent ($x'$ is a permutation of the byte ordering of $x$ without fixed point). Our simulations indicate that such a selection function with XOR and modular addition imbricated is roughly equivalent to a modular addition one.

Finally, going backward in the computation to the first $P_C$ layer, the selection function for the fourth step is of XOR type and recovers $cv_3$ byte per byte:

$$y[b] = (cv_3[b] \boxplus cv_1[b]) \oplus w'.$$

Note that each step must be successful, otherwise it would compromise the results of the following steps. Overall, we are able to recover the full chaining variable $CV$ by performing 64 byte-wise XOR correlation analysis and 64 byte-wise modular addition correlation analysis.
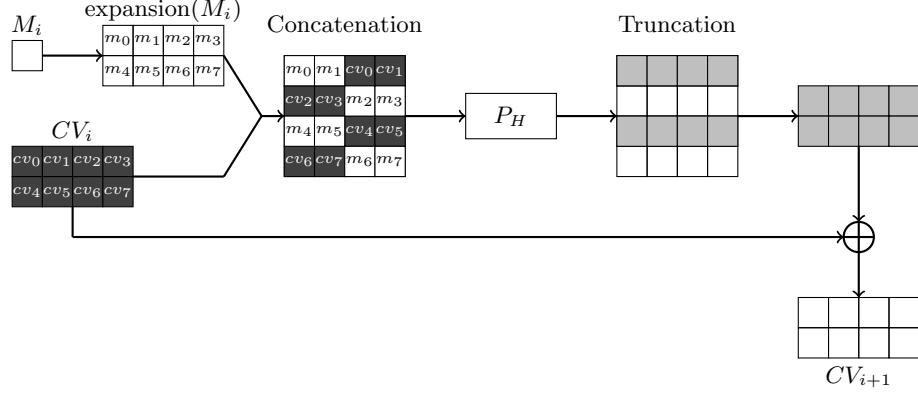
**Fig. 11.** The compression function of `HAMSI`. Each cell represents a 32-bit word.

### 5.3 HAMSI

**Description.** `HAMSI` [36] is an iterated hash function whose compression function $h$ maps a 256-bit chaining variable $CV_i$ and a 32-bit message block $M_i$ to a new chaining variable $CV_{i+1}$. First, the message block is expanded into eight 32-bit words $m_i$ that are concatenated to $CV_i$ in order to initialize a 512-bit internal state (viewed as a $4 \times 4$ matrix of 32-bit words). Then a permutation $P_H$ is applied to the state and a truncation allows to extract 256 bits. In the end, there is a feedforward of the chaining variable (see Figure 11):

$$CV_{i+1} = trunc(P_H(CV_i||expansion(M_i))) \oplus CV_i.$$

The permutation $P_H$ contains three identical rounds. One round is composed of three layers: constants are first XORed to the internal state, then 4-bit Sboxes are applied to the whole state by taking one bit of each 32-bit word of the same column of the $4 \times 4$ matrix and repeating the process for all bit positions. Finally, a linear layer is applied to four 32-bit words diagonals of the $4 \times 4$ matrix independently (see Figure 12).
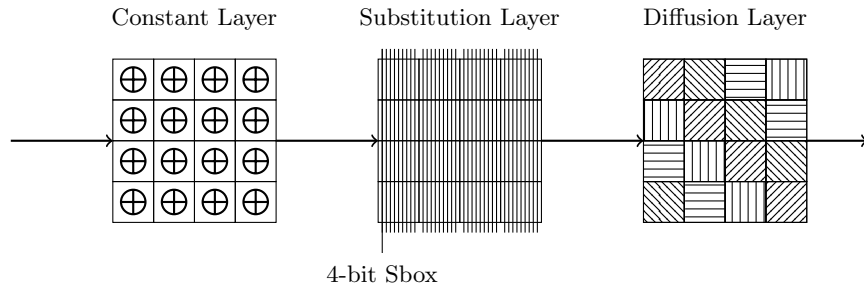


**Fig. 12.** One round of the internal permutation $P_H$ of `HAMSI`. Each cell represents a 32-bit word.

**Side-channel analysis.** The known random message $M$ (after expansion) and the secret chaining variable $CV$ fill the internal state matrix as shown in Figure 11 (32-bit words are denoted $m_i$ and $cv_i$). The first layer of the permutation $P_H$ XORs each matrix element with a constant $t_i$:

$$m'_i = m_i \oplus t_i \text{ and } cv'_i = cv_i \oplus t_{i+8}.$$

Then is applied the `HAMSI` Sbox layer. This Sbox is acting over 4-bits (one bit per word located in the same column of the state matrix). Therefore, the input of each Sbox is composed of 2 known message bits and 2 unknown chaining variable bits. The generic selection function for `HAMSI` can therefore be written as:

$$w = Sbox(m'_i[b]||cv'_{i+2}[b]||m'_{i+4}[b]||cv'_{i+6}[b]) \text{ or } w = Sbox(cv'_i[b]||m'_{i+2}[b]||cv'_{i+4}[b]||m'_{i+6}[b])$$

for $i \in [0,1]$ and $b \in [0,\ldots,127]$ where b is the bit index in a 128-bits word. Overall, one recovers two bits of $cv'_i$ at a time with a total of 4 times 128 correlation computations (with 4 guess each).

## 6   Conclusion and Discussions

For each hash proposal considered in this article, we described an appropriate selection function for an efficient side-channel attack. The table below summarize these results in terms of selection function and secret chaining variable size.

| Candidates | Selection function | Correlation analysis |
|---|---|---|
| ECHO | AES Sbox | 64 analysis at byte level (x4 possibilities) |
| Grøstl | AES Sbox | 64 analysis at byte level |
| SHAvite-3 | AES Sbox | 16 + 16 analysis at byte level |
| BLAKE | Modular addition | 32 analysis at byte level |
| CubeHash | Modular addition and XOR | 64 ADD + 64 XOR analysis at byte level |
| HAMSI | HAMSI Sbox | 128 analysis at 2-bit level |

In the case of the `AES`-based `SHA-3` candidates, we did not found significant differences of performance when choosing the selection function. Indeed, in `ECHO`, `Grøstl` and `SHAvite-3`, one has to compute several `AES` Sbox side-channel attacks in order to retrieve the full secret internal state. Up to a small complexity/number of power traces factor, the three schemes seem to provide the same natural vulnerability to side-channel cryptanalysis. As expected, their situation is therefore very close to the real `AES` block cipher.

Attacking `BLAKE` seems feasible in practice since we managed to derive a modular addition selection function for recovering the 256 bits of unknown chaining variable. The modular addition non-linearity is very valuable for the attacker as it increases the correlation contrast. Then, for `CubeHash` (a typical ARX function) we tried to force the selection function to be of modular addition type as much as possible. Overall, 512 bits can be recovered with modular addition selection function and 512 bits with XOR selection function. In practice, depending on the underlying hardware, it could be challenging to mount the attack. One must notice that the internal state is bigger for `CubeHash` than for other candidates. This is an additional strength since in practice, if the side-channel attack gives only probabilistic results, the final exhaustive search complexity will be higher. Finally, for `HAMSI`, the attack would be difficult to mount despite the fact that a substitution table is used. Indeed, the correlation contrast for this primitive is quite low compared to the `AES` Sbox. We believe that a better selection function involving a modular addition might possibly be found in the inner layer.

Of course, those results concern unprotected implementations and the ranking would be really different if we also considered methods for hardening the side-channel cryptanalysis.

For example, in the case of `AES`-based hash functions, one could perform secure round computations and leverage all the research achieved so far on this subject [2, 21]. Also, as `ECHO` processes parallel `AES` rounds, we believe it could benefit from secure bit-slice implementations regarding some side-channels attacks [27], while maintaining its normal use efficiency. Finally, `ECHO` and `SHAvite-3` can take advantage of the natural protection inherited from the hardware implementations of the `AES` round such as the new AES NI instruction set on Intel microprocessor [8].

Side-channel countermeasures for ARX constructions such as `BLAKE` or `CubeHash` are of course possible, but they will require to constantly switch from boolean to arithmetic masking. As a consequence, one will observe an important decrease of the speed performance for secure implementations. `AES`-based hash functions seem naturally easier to attack regarding side-channel cryptanalysis, but are also easier to protect.

# References

1. Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM Side-Channel(s). In Jr. et al. [23], pages 29–45.
2. Mehdi-Laurent Akkar and Christophe Giraud. An Implementation of DES and AES, Secure against Some Attacks. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES*, volume 2162 of *LNCS*, pages 309–318. Springer, 2001.
3. Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST, 2008.
4. Mihir Bellare. New Proofs for NMAC and HMAC: Security Without Collision-Resistance. Cryptology ePrint Archive, Report 2006/043, 2006. `http://eprint.iacr.org/`.
5. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *LNCS*, pages 1–15. Springer, 1996.
6. Mihir Bellare, Joe Kilian, and Phil Rogaway. The Security of Cipher Block Chaining. In *Advances in Cryptology -CRYPTO'94*, Santa Barbara, Californie, tats-Unis, pages 341–355. LNCS 839, Springer-Verlag, 1994.
7. Ryad Benadjila, Olivier Billet, Henri Gilbert, Gilles Macario-Rat, Thomas Peyrin, Matt Robshaw, and Yannick Seurin. SHA-3 Proposal: ECHO. Submission to NIST, 2008. Available online at `http://crypto.rd.francetelecom.com/echo/`.
8. Ryad Benadjila, Olivier Billet, Shay Gueron, and Matthew J. B. Robshaw. The Intel AES Instructions Set and the SHA-3 Candidates. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 162–178. Springer, 2009.
9. Daniel J. Bernstein. CubeHash specification (2.B.1). Submission to NIST (Round 2), 2009.
10. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Radiogatun, a belt-and-mill hash function. Presented at Second Cryptographic Hash Workshop, Santa Barbara (August 24-25, 2006). See `http://radiogatun.noekeon.org/`.
11. Eli Biham and Orr Dunkelman. The SHAvite-3 Hash Function. Submission to NIST (Round 2), 2009. `http://www.cs.technion.ac.il/~orrd/SHAvite-3/Spec.15.09.09.pdf`.
12. Gilles Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *LNCS*. Springer, 1990.
13. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Joye and Quisquater [22], pages 16–29.
14. Ivan Damgård. A Design Principle for Hash Functions. In Brassard [12], pages 416–427.
15. FIPS 197. Advanced Encryption Standard. Federal Information Processing Standards Publication 197, 2001. U.S. Department of Commerce/N.I.S.T.
16. FIPS 46-3. Data Encryption Standard. Federal Information Processing Standards Publication, 1999. U.S. Department of Commerce/N.I.S.T.
17. Pierre-Alain Fouque, Gaëtan Leurent, Denis Réal, and Frédéric Valette. Practical Electromagnetic Template Attack on HMAC. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2009.
18. Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a SHA-3 candidate. Submission to NIST, 2008. Available online at `http://www.groestl.info`.

19. Praveen Gauravaram and Katsuyuki Okeya. An Update on the Side Channel Cryptanalysis of MACs Based on Cryptographic Hash Functions. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *INDOCRYPT*, volume 4859 of *Lecture Notes in Computer Science*, pages 393–403. Springer, 2007.

20. Praveen Gauravaram and Katsuyuki Okeya. Side Channel Analysis of Some Hash Based MACs: A Response to SHA-3 Requirements. In Liqun Chen, Mark Dermot Ryan, and Guilin Wang, editors, *ICICS*, volume 5308 of *Lecture Notes in Computer Science*, pages 111–127. Springer, 2008.

21. Jovan Dj. Golic and Christophe Tymen. Multiplicative Masking and Power Analysis of AES. In Jr. et al. [23], pages 198–212.

22. Marc Joye and Jean-Jacques Quisquater, editors. *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *LNCS*. Springer, 2004.

23. Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *LNCS*. Springer, 2003.

24. John Kelsey. How to Choose SHA-3. `http://www.lorentzcenter.nl/lc/web/2008/309/presentations/Kelsey.pdf`.

25. Lars R. Knudsen, Christian Rechberger, and Søren S. Thomsen. The Grindahl Hash Functions. In Alex Biryukov, editor, *FSE*, volume 4593 of *LNCS*, pages 39–57. Springer, 2007.

26. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO*, pages 388–397, 1999.

27. Robert Könighofer. A Fast and Cache-Timing Resistant Implementation of the AES. In *CT-RSA*, pages 187–202, 2008.

28. Kerstin Lemke, Kai Schramm, and Christof Paar. DPA on n-Bit Sized Boolean and Arithmetic Operations and Its Application to IDEA, RC6, and the HMAC-Construction. In Joye and Quisquater [22], pages 205–219.

29. Robert P. McEvoy, Michael Tunstall, Colin C. Murphy, and William P. Marnane. Differential Power Analysis of HMAC Based on SHA-2, and Countermeasures. In *WISA*, pages 317–332, 2007.

30. Ralph C. Merkle. One Way Hash Functions and DES. In Brassard [12], pages 428–446.

31. National Institute of Standards and Technology. FIPS 180-1: Secure Hash Standard. `http://csrc.nist.gov`, April 1995.

32. National Institute of Standards and Technology. FIPS PUB 197, Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, U.S. Department of Commerce, 2001.

33. National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a NewCryptographic Hash Algorithm (SHA-3) Family. *Federal Register*, 27(212):62212–62220, November 2007. Available:`http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf`(2008/10/17).

34. NIST. FIPS 198 – The Keyed-Hash Message Authentication Code (HMAC), 2002.

35. Katsuyuki Okeya. Side Channel Attacks Against HMACs Based on Block-Cipher Based Hash Functions. In Lynn Margaret Batten and Reihaneh Safavi-Naini, editors, *ACISP*, volume 4058 of *Lecture Notes in Computer Science*, pages 432–443. Springer, 2006.

36. Özgül Kücük. The Hash Function Hamsi. Submission to NIST (updated), 2009.

37. Ronald L. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. `http://www.ietf.org/rfc/rfc1321.txt`, April 1992.

38. Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In *CRYPTO*, pages 17–36, 2005.

39. Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In *EUROCRYPT*, pages 19–35, 2005.