Differential Power Analysis of HMAC based on SHA-2, and Countermeasures

Robert McEvoy, Michael Tunstall, Colin C. Murphy, and William P. Marnane

Department of Electrical & Electronic Engineering, University College Cork, Ireland
{robertmce,miket,cmurphy,liam}@eleceng.ucc.ie

Abstract. The HMAC algorithm is widely used to provide authentication and message integrity to digital communications. However, if the HMAC algorithm is implemented in embedded hardware, it is vulnerable to side-channel attacks. In this paper, we describe a DPA attack strategy for the HMAC algorithm, based on the SHA-2 hash function family. Using an implementation on a commercial FPGA board, we show that such attacks are practical in reality. In addition, we present a masked implementation of the algorithm, which is designed to counteract first-order DPA attacks.

1 Introduction

In today's modern society of e-mail, internet banking, online shopping and other sensitive digital communications, cryptography has become a vital tool for ensuring the privacy of data transfers. To this end, Message Authentication Code (MAC) algorithms are used to verify the identity of the sender and receiver, and to ensure the integrity of the transmitted message. These algorithms process the message to be authenticated along with a secret key, which is shared between the sender and receiver. The result is a short string of bits, called a MAC. HMAC [1] is a popular type of MAC algorithm which is used in the IPsec [14] and TLS protocols [6], and is based on a cryptographic hash function such as SHA-2 [16].

The last decade has also seen the emergence of attacks which target cryptographic algorithms that are implemented in embedded hardware [9]. Of particular interest are differential side-channel attacks, such as Differential Power Analysis (DPA) [12]. These non-invasive attacks exploit information that leaks from a cryptographic device via some side channel, such as timing information, power consumption, or electromagnetic emanations. Comparing small variations in the side-channel information as a device processes different messages can potentially allow an attacker to recover secret information stored within the device. In this paper, we examine the susceptibility to differential side-channel attacks of the HMAC algorithm based on the SHA-2 family of hash functions.

Side-channel attacks on hash functions and the HMAC algorithm have been discussed in the past, but specific attack details for the SHA-2 family have not been given, nor have countermeasures been designed. In 2001, Steinwandt et al. [21] presented a theoretical attack on the SFLASH signature scheme, which

targeted an exclusive-OR (XOR) operation in SHA-1. Coron and Tchoulkine [5] noted the vulnerability of the HMAC algorithm to a DPA attack. Lemke et al. [10] described a theoretical DPA attack on the HMAC algorithm based on the hash function RIPEMD-160, noting that a similar approach could be taken for a HMAC scheme based on SHA-1. Okeya et al. [18, 19] highlight the susceptibility of MAC and HMAC algorithms to side-channel attacks, but the exposition is for the HMAC algorithm based on block-cipher based hash functions, in contrast with SHA-2, which is a dedicated cryptographic hash function.

In this paper, we characterise a differential side-channel attack on an implementation of the HMAC algorithm that uses the SHA-2 hash function family. Furthermore, we provide attack results on a FPGA implementation of the algorithm. We also describe countermeasures that could be used to prevent such side-channel attacks, by designing masked circuits for the vulnerable SHA-2 operations. The rest of this paper is organised as follows. In Section 2, the necessary background theory regarding the HMAC algorithm, the SHA-2 family, and DPA attacks is introduced. Section 3 gives a detailed account of how the SHA-256 based HMAC scheme can be broken by a side-channel attacker. Results from a practical FPGA-based implementation of this attack are presented in Section 4. In Section 5, a masking scheme is designed as a countermeasure against the attack, and the resulting FPGA-based scheme is tested in Section 6. Section 7 concludes the paper.

2 Background Theory

2.1 HMAC Algorithm Overview

The HMAC authentication scheme was first introduced by Bellare et al. at CRYPTO'96 [1]. The scheme was designed such that the security of the MAC is built upon the security of the underlying hash function h. The MAC is calculated as follows:

$$\mathrm{HMAC}_k(m) = h((k \oplus opad)||h((k \oplus ipad)||m)) \tag{1}$$

where k is the secret key (padded with zeros to equal the block size of h), and m is the message to be authenticated. *ipad* is a fixed string whose length equals the block size of h; generated by repeating the hexadecimal byte 0x36. Similarly, *opad* is fixed and is formed by repeating the hexadecimal byte 0x5C. $\oplus$ and $||$ denote XOR and concatenation respectively.

Clearly, in order to calculate $\mathrm{HMAC}_k(m)$, the hash function h must be invoked twice. In this paper, we focus on the first call to the hash function, which calculates the partial MAC:

$$\mathrm{HMAC}'_k(m) = h((k \oplus ipad)||m) \tag{2}$$

In [1], the authors suggested using MD5 or SHA-1 to instantiate the hash function h. In 2002, the HMAC algorithm was released as a standard by NIST [17], in which h is defined as a NIST-approved hash function. In this paper, we adhere to this standard and choose SHA-256 to instantiate h. This follows a recent

trend in the cryptographic community away from older hash functions, for which weaknesses have been identified [11], and towards newer constructions like the SHA-2 family [16]. We use the term "HMAC-SHA-256" to denote the HMAC algorithm that uses SHA-256 to instantiate h.

2.2 SHA-256 Description

There are four hash functions in the SHA-2 family: SHA-224, SHA-256, SHA-384 and SHA-512. Each algorithm generates a fixed-length hash value; SHA-224 produces a 224-bit output, SHA-256 has a 256-bit output, etc. The compression functions in SHA-224 and SHA-256 are based on 32-bit operations, whereas the compression functions for SHA-384 and SHA-512 are based on 64-bit operations. We focus on SHA-256 in our attacks, because it is easier in practice to perform a side-channel attack on a 32-bit word than on a 64-bit word. However, in theory, the side-channel attacks and countermeasures described in this paper should also be applicable to HMAC-SHA-384 and HMAC-SHA-512.

The SHA-256 algorithm essentially consists of three stages: (i) message padding and parsing; (ii) expansion; and (iii) compression.

Message Padding and Parsing The binary message to be processed is appended with a '1' and padded with zeros until its bit length $\equiv 448 \bmod 512$. The original message length is then appended as a 64-bit binary number. The resultant padded message is parsed into N 512-bit blocks, denoted $M^{(i)}$, for $1 \leq i \leq N$. These $M^{(i)}$ message blocks are passed individually to the message expansion stage.

Message Expansion The functions in the SHA-256 algorithm operate on 32-bit words, so each 512-bit $M^{(i)}$ block from the padding stage is viewed as sixteen 32-bit blocks denoted $M_t^{(i)}$, $1 \leq t \leq 16$. The message expansion stage (also called the message scheduling stage) takes each $M^{(i)}$ and expands it into sixty-four 32-bit W_t blocks for $1 \leq t \leq 64$, according to equations given in [16].

Message Compression The W_t words from the message expansion stage are then passed to the SHA compression function, or the 'SHA core'. The core utilises eight 32-bit working variables labelled A, B, ..., H, which are initialised to predefined values $H_0^{(0)}$–$H_7^{(0)}$ (given in [16]) at the start of each call to the hash function. Sixty-four iterations of the compression function are then performed, given by:

$$A = T1 \boxplus T2 \quad (3) \qquad E = D \boxplus T1 \quad (7)$$

$$B = A \quad (4) \qquad F = E \quad (8)$$

$$C = B \quad (5) \qquad G = F \quad (9)$$

$$D = C \quad (6) \qquad H = G \quad (10)$$

where

$$T1 = H \boxplus \sum\nolimits_1(E) \boxplus Ch(E,F,G) \boxplus K_t \boxplus W_t \tag{11}$$

$$T2 = \sum\nolimits_0(A) \boxplus Maj(A,B,C) \tag{12}$$

$$Ch(x,y,z) = (x \ \wedge \ y) \oplus (\bar{x} \ \wedge \ z) \tag{13}$$

$$Maj(x,y,z) = (x \ \wedge \ y) \oplus (x \ \wedge \ z) \oplus (y \ \wedge \ z) \tag{14}$$

$$\sum\nolimits_0(x) = ROT_2(x) \oplus ROT_{13}(x) \oplus ROT_{22}(x) \tag{15}$$

$$\sum\nolimits_1(x) = ROT_6(x) \oplus ROT_{11}(x) \oplus ROT_{25}(x) \tag{16}$$

and the inputs denoted K_t are 64 32-bit constants, specified in [16]. All additions in the SHA-256 algorithm are computed modulo 2^{32}, denoted by $\boxplus$. The logical AND operator is denoted by $\wedge$, and $\bar{x}$ denotes the logical NOT operator. After 64 iterations of the compression function, a 256-bit intermediate hash value $H^{(i)}$, comprising $H_0^{(i)}$–$H_7^{(i)}$, is calculated:

$$H_0^{(i)} = A \boxplus H_0^{(i-1)}, H_1^{(i)} = B \boxplus H_1^{(i-1)}, \ldots, \ H_7^{(i)} = H \boxplus H_7^{(i-1)} \tag{17}$$

The SHA-256 compression algorithm then repeats and begins processing another 512-bit block from the message padder. After all N data blocks have been processed, the output, $H^{(N)}$, is formed by concatenating the final hash values:

$$H^{(N)} = H_0^{(N)} \ || \ H_1^{(N)} \ || \ H_2^{(N)} \ || \ \ldots \ || \ H_7^{(N)} \tag{18}$$

2.3 Differential Side-Channel Analysis

Some of the most common forms of side-channel analysis are Differential Power Analysis (DPA) [9] and related attacks such as Correlation Power Analysis (CPA) [2]. In this type of attack, a series of power consumption traces are acquired using an oscilloscope, where each trace has a known associated input (e.g. the message block being processed). A comprehensive guide to this class of attacks is provided in [12].

The fundamental principle behind all DPA attacks is that at some point in an algorithm's execution, a function f exists that combines a fixed secret value with a variable which an attacker knows. An attacker can form hypotheses about the fixed secret value, and compute the corresponding output values of f by using an appropriate leakage model, such as the Hamming Distance model [2]. The attacker can then use the acquired power consumption traces to verify her hypotheses, by partitioning the acquisitions or using Pearson's correlation coefficient. These side-channel analysis attacks are aided by knowledge of details of the implementation under attack. Moreover, these attacks can be used to validate hypotheses about implementation details. In subsequent sections, these side-channel analysis attacks are referred to as DPA attacks.

3 Attacking HMAC-SHA-256

In this section, we describe an attack on HMAC-SHA-256 using DPA. This attack does not allow recovery of the secret key itself, but rather a secret intermediate state of the SHA-256 hash function. Knowledge of this intermediate state would allow an attacker to forge MACs for arbitrary messages. We note that the attack is not limited to DPA, and other side-channels, such as the electromagnetic side-channel, could also be used.

3.1 Goal of the attack

We assume that the attacker has access to a device that performs the HMAC algorithm, and that she has knowledge of the messages being processed by the device. This target device contains a basic implementation of the SHA-256 algorithm, and does not include any side-channel analysis countermeasures. Furthermore, we assume that the attacker has access to some side-channel information (e.g. the power consumption) while the device is calculating the MAC, which leaks the Hamming Distance between the internal signals as they change from one state to the next. As is common, we assume that the secret key is stored in a secure environment, which does not leak side-channel information.

The attack focuses on the first execution of SHA-256, given in equation (2). The block size of SHA-256 is 512 bits; therefore, using the notation from Section 2.2, $|k| = |ipad| = 512$. Without loss of generality, we can assume that the size of the message m is such that $N = 2$, i.e. the device will run through the 64 iterations of the compression function twice, in order to calculate equation (2).

When $i = 1$, the hash function is operating on $(k \oplus ipad)$, which clearly does not change from one execution of the HMAC algorithm to the next. Hence, the intermediate hash $H^{(1)}$ is also fixed and unknown. Recall that in order to perform a differential side-channel attack, we require fixed unknown data to be combined with variable known data. This criterion is fulfilled during the calculation of $H^{(2)}$, when the variable m is introduced and combined with the previous intermediate hash $H^{(1)}$. Therefore, in theory, a differential side-channel attack could be launched on a device calculating equation (2), in order to recover $H^{(1)}$. This knowledge would allow the attacker to create partial MACs of her choice. Reapplying the side-channel attack on the second invocation of SHA-256 in the HMAC algorithm would allow the attacker to forge full MACs for messages of her choosing. Consequently, the goal of the attacker is to recover the secret intermediate hash value $H^{(1)}$.

3.2 Attack strategy

The secret intermediate hash $H^{(1)}$ manifests itself as the initial values of the eight 32-bit working variables A–H, when $i = 2$. We use the subscript $t, 1 \leq t \leq 64$ to denote the round number, e.g. A_1 refers to the value of A at the start of round 1 of the compression function, etc. The side-channel attacker's goal is to uncover the eight variables A_1–H_1. A strategy for such an attack is now described.

1. With reference to equations (3) and (7), it is clear that at some point in the first round, the variable $T1_1$ must be calculated. $T1_t$ is a large sum with 5 terms, and can be re-written as:

$$T1_t = \theta_t \boxplus W_t \tag{19}$$

where

$$\theta_t = H_t \boxplus \sum\nolimits_1(E_t) \boxplus Ch(E_t, F_t, G_t) \boxplus K_t \tag{20}$$

In round 1, θ_1 is fixed and unknown, and W_1 is known by the attacker, since it is related to m. Therefore, a DPA attack can be launched by making hypotheses about θ_1, and computing the corresponding values of $T1_1$. Since SHA-256 uses 32-bit words, 2^{32} hypotheses for θ_1 are required. Furthermore, since we assume that the target device leaks the Hamming Distance (HD), the 2^{32} possibilities for the previous state, $T1_0$, must also be taken into account. Therefore, the attacker correlates the power traces with her 2^{64} hypotheses for $\text{HD}(T1_0, T1_1)$. This allows the attacker to recover $T1_0$ and θ_1, and then calculate $T1_1$ for any message m.
Clearly, correlating with 2^{64} hypotheses would be computationally infeasible, even for well-resourced attackers. In Section 3.3, we describe how the Partial CPA technique [22] can be used to significantly reduce the attack's complexity.
2. The above attack stage gives the attacker control over the value of $T1_1$, so it is now a known variable. Using equation (7), the attacker can now make hypotheses on the (fixed) bits of D_1, using the bits of E_2 as selection bits. Using the Hamming Distance model, hypotheses for the previous (secret) state E_1 are also generated. In this way, the attacker can recover her first secrets, D_1 and E_1, and accurately predict the value of E_2 for any message m.
3. Focusing on equation (3), we observe that $T1_1$ is variable and known, whereas $T2_1$ is fixed and unknown. The attacker can launch a DPA attack on A_2 by forming hypotheses about $T2_1$ and the previous state A_1. Hence, the secret value of A_1 is revealed. Furthermore, with knowledge of both $T1_1$ and $T2_1$, the attacker can now accurately predict A_2 for any message m.
Therefore, by analysing the side-channel signals from the first round, the attacker can recover the fixed secret values of θ_1, D_1, E_1, $T2_1$ and A_1, and also predict the values of variables $T1_1$, A_2 and E_2.
4. The attacker now turns her attention to the second SHA-256 round. Here, the Ch function is calculated as:

$$Ch(E_2, F_2, G_2) = (E_2 \ \wedge \ F_2) \oplus (\overline{E_2} \ \wedge \ G_2) \tag{21}$$

where E_2 is variable, and known by the attacker. From equations (8) and (9), we observe that F_2 and G_2 are fixed at E_1 and F_1, respectively. Therefore, the attacker can generate hypotheses about the unknown values F_1, and attack the output of the Ch function. Of course, 2^{32} hypotheses for the previous state $Ch(E_1, F_1, G_1)$ are also required. Recovering F_1 means that the attacker can now accurately predict the variable $Ch(E_2, F_2, G_2)$.

5. The next point of attack is the calculation of $T1_2$ (equation (11)). At this stage, the only fixed unknown value in the equation is H_2, as every other variable can be predicted. The attacker already knows the previous state $T1_1$ from stage 1 above. Mounting a DPA attack uncovers H_2, and allows $T1_2$ to be predicted. From equation (10), it can be seen that H_2 is equivalent to G_1.
6. The knowledge of $T1_2$ gained from the previous attack stage can be applied to equation (7). Using the bits of E_3 as the selection function, the attacker can mount a DPA attack that uncovers D_2. From equation (6), we observe that D_2 is equivalent to C_1.
7. The Maj function in the second round can be expressed as:

$$Maj(A_2, B_2, C_2) = (A_2 \ \wedge \ B_2) \oplus (A_2 \ \wedge \ C_2) \oplus (B_2 \ \wedge \ C_2) \qquad (22)$$

   where A_2 is variable, and known by the attacker. From equations (4) and (5), we observe that B_2 and C_2 are fixed at A_1 and B_1, respectively. Using a similar approach to that taken in stage 4 above, the attacker can perform DPA on Maj and discover the secret value of B_1.
8. By following the above strategy, the attacker can recover the fixed secrets A_1–G_1. The last remaining secret variable, H_1, can be found by reverting the focus to round 1, and substituting into equation (11), where the only unknown value is that of H_1. The eight 32-bit secret values are thus recovered, using seven first-order DPA attacks.

3.3 Complexity of the attack

As noted above, it is currently computationally infeasible for an attacker to compute 2^{64} hypotheses for a DPA attack. However, as indicated in [2] and illustrated in [22], a partial correlation may be computed, rather than the full correlation. If a correlation coefficient of ρ is obtained by correctly predicting all 32 bits of an intermediate variable, then we would expect to obtain a partial correlation of $\rho\sqrt{l/32}$ by predicting l bits correctly. Hypotheses can be made on smaller sets of bits at a time, e.g. $l = 8$, and this strategy can be employed to keep only those hypotheses that produce the highest partial correlations. In this way, the full 32-bit correlation can be built up in stages, thereby reducing the complexity of the attack. This is similar to the 'extend-and-prune' strategy employed by a template attack [3].

4 Attack on FPGA Implementation

4.1 Implementation Details

In order to demonstrate the feasibility of a DPA attack on HMAC-SHA-256, we implemented the algorithm on a Xilinx FPGA Board. FPGAs are attractive for implementing cryptographic algorithms because of their low cost (relative to ASICs), and their flexibility when adopting security protocol upgrades. FPGAs

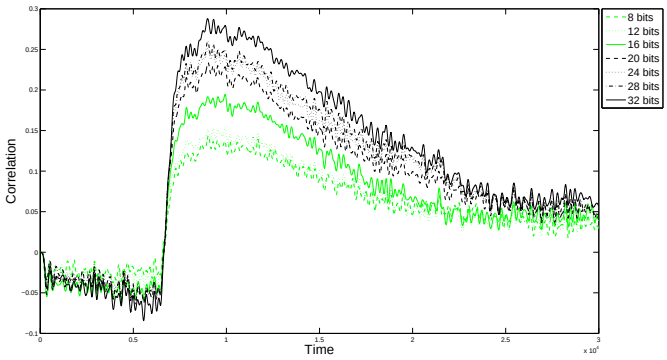

Fig. 1. Correlation and partial correlations between the power consumption and E_2, given the correct prediction of D_1 and the previous state E_1.

also allow rapid prototyping of various designs. For our experiments, we implemented SHA-256 on Xilinx's low-cost Spartan™-3E Development Kit, which contains the Spartan™XC3S500E FPGA [23]. FPGAs consist mostly of Configurable Logic Blocks (CLBs), arranged in a regular array across the chip. In our case, each CLB contains 4 logic "slices", which is the basic unit used to quantify the area occupied by a design. Each slice contains two four-input Look-Up Tables (LUTs) and two registers. Logic also exists within a slice which allows fast implementation of carry look-ahead addition. Each slice has a dedicated multiplexer, which is hard-wired to provide a fast carry chain between consecutive slices and CLBs. Indeed, fast carry logic is a feature of many modern FPGAs. We will make use of this dedicated addition circuitry in Section 5.

Several optimisations for the SHA-2 family exist, such as pipelining and loop unrolling [4]. However, for simplicity, it was decided to implement a basic design without such optimisations. The design was captured using VHDL, and synthesis, placing and routing were performed using Xilinx ISE™v9.1i. The processor and interface circuitry utilise 951 slices, corresponding to 20% of the FPGA resources. The critical path in the design (i.e. the longest combinational path) is 16.4 ns, during the calculation of A (equation (3)). Block RAMs (BRAMs) on the FPGA are used to store the various messages to be processed; this reduces the communication requirements with the FPGA board.

4.2 Experimental Results

In order to obtain DPA power traces from the design, the FPGA board was configured with the basic SHA-256 design, and a 10 Ω resistor was inserted in the FPGA power supply line. Using a LeCroy WaveRunner 104Xi oscilloscope and a differential probe, we could measure the voltage fluctuations across the resistor as the SHA-256 algorithm was being executed. Therefore, the traces recorded on

the oscilloscope were proportional to the power consumption of the FPGA during the execution of the algorithm. Traces for the first three rounds were captured while 4000 random messages were being processed by the FPGA. In order to reduce acquisition noise, each captured trace corresponded to the average of 700 executions with the same message. Figure 1 shows the correlation trace achieved when D_1 and the unknown previous state E_1 are correctly predicted. The different levels correspond to the correlation coefficients achieved when a certain number of bits are correctly predicted.

5 Masking the SHA-256 algorithm

The preceding sections have demonstrated the susceptibility of hardware implementations of the HMAC algorithm to first-order DPA attacks. We now examine how to use masking as a countermeasure to such attacks. The masking technique aims to use random values to conceal intermediate variables in the implementation of the algorithm, thereby making the side-channel leakage independent of the secret intermediate variables. Much of the literature has focused on masking techniques for software implementations of cryptographic algorithms (e.g. [5, 8, 15]). In [7], Golić detailed techniques for masking hardware implementations, which we build upon below in order to mask HMAC-SHA-256.

5.1 Requirements

Consider a function f and intermediate variables x, y and z, such that $z = f(x, y)$. If x or y are key-dependent or data-dependent, then masking is required. We introduce random masks r_x, r_y and r_z such that $x' = x \circ r_x$, $y' = y \circ r_y$ and $z' = z \diamond r_z$; where $\circ$ is the group operation masking the input data, and $\diamond$ is the group operation masking the output z. In order to prevent differential side-channel attacks, the function f must be modified to the function f', such that $z' = f'(x', y', r_x, r_y, r_z) = z \diamond r_z$. If the group operation is XOR, the masking scheme is termed Boolean masking. The SHA-256 algorithm also uses addition modulo 2^{32}, which requires arithmetic masking.

In [7], Golić described the goal of designing a secure masked hardware implementation for a function f, using the "secure computation condition". This condition states that the output value of each logic gate in the design should be statistically independent of the original data (i.e. the secret key and the input data). In the case of a Boolean logic circuit implementing f, this condition is satisfied if all of the inputs to the circuit are jointly statistically independent of the original data. In the case of a multiplexer-based design for f: (i) the data inputs to the multiplexer should be identically distributed; (ii) each data input should be statistically independent of the original data; and (iii) for each fixed value of the original data, each data input should be statistically independent of the control input. In order to mask our SHA-256 design correctly, care must be taken that these conditions are met.

Table 1. Linear and non-linear functions (with respect to XOR) used in SHA-256

Linear	Non-Linear
NOT ($\bar{x}$)	Ch
σ_0	Maj
σ_1	AND
$\sum_0$, $\sum_1$	addition modulo 2^{32}

5.2 Masking the original data

In Section 3, the eight variables A–H in the SHA-256 algorithm were identified as the secret values which are of interest to the side-channel attacker. Therefore, we begin the first iteration of the compression function by XOR-ing these values with eight 32-bit random masks denoted r_A–r_H, so that they become A'–H'. Furthermore, the variable input data W_t to the SHA-256 compression algorithm requires masking. Since this data is perfectly predictable by the attacker, it must be XOR-ed with a new 32-bit random value r_W in every SHA-256 round.

Recall that the fixed secret data mixes with the attacker's variable known data within the SHA-256 compression algorithm. Therefore, we must also mask the individual functions in the SHA-256 compression algorithm. If a function f is linear with respect to the mask, then it is easy to mask, as $z' = f(x', y')$, and $r_z = f(r_x, r_y)$. Conversely, non-linear functions require modification in order to achieve secure masking. Therefore, new circuits implementing these non-linear functions must be designed, with respect to the secure computation condition given above. Table 1 outlines the linear and non-linear functions used by SHA-256. In the following sub-sections, we present our designs for the secure circuits implementing the non-linear functions of Table 1 on an FPGA.

5.3 The Ch and Maj Functions

The logical functions Ch and Maj are described by equations (13) and (14) respectively. The non-linearity in both of these functions stems from the AND operations. Therefore, Ch and Maj cannot be implemented using ordinary AND gates, and masked AND gates must be used instead.

In [7], Golić proposed masking the AND function $z = x \wedge y$, using Boolean masking, as follows:

$$\begin{aligned} z' &= \wedge'(x', y', r_x, r_y)) \\ &= \overline{y'} \wedge (\overline{r_y} \wedge r_x \vee r_y \wedge x') \vee y' \wedge (r_y \wedge r_x \vee \overline{r_y} \wedge x') \end{aligned} \tag{23}$$

where $\vee$ denotes logical OR. This approach has the advantage that the output mask r_z is equal to the input mask r_x; therefore, a new mask for z is not required.

When implementing this masked AND circuit (denoted $\wedge'$) on an FPGA, we can take advantage of the underlying slice structure. Equation (23) is a four-input function, which is perfectly suited for implementation in one of the FPGA slice's four-input LUTs. A two-input or three-input XOR operation can also be

implemented using a single four-input LUT. Note that care must also be taken when describing masked circuits, so that the HDL synthesis tool does not remove the redundancy in the design, or combine two variables that are not statistically independent.[1]

Our design for masked $Maj(A, B, C)$, denoted $Maj'(A', B', C', r_A, r_B, r_C)$, is as follows:

$$Maj'(A', B', C', r_A, r_B, r_C) = (\wedge'(A', B', r_A, r_B)) \oplus (\wedge'(A', C', r_A, r_C)) \\ \oplus(\wedge'(B', C', r_B, r_C)) \quad (24)$$

Therefore, three LUTs (per bit) are required for the three masked AND operations, and one LUT (per bit) is required for the three-input XOR. Since the variables are 32-bit, Maj' requires 128 LUTs or 64 Spartan-3E slices. This is four times larger than a basic unmasked implementation of Maj. By choosing the order for the operands of the masked AND functions appropriately, the output mask becomes $r_{Maj} = r_A \oplus r_A \oplus r_B = r_B$.

Similarly, our design for masked $Ch(E, F, G)$ is

$$Ch'(E', F', G', r_E, r_F, r_G) = (\wedge'(E', F', r_E, r_F)) \oplus (\wedge'(G', \overline{E'}, r_G, r_E)) \quad (25)$$

which requires two LUTs (per bit) for the two masked AND operations, and one LUT (per bit) for the 2-input XOR. Care must be taken regarding the order of the operands of the masked AND functions. If E' was the first operand of both masked AND gates, then the output mask would be $r_{Ch} = r_E \oplus r_E = 0$, i.e. the output would be unmasked. Therefore, we choose the order such that $r_{Ch} = r_E \oplus r_G$, which requires one extra LUT (per bit) to compute the XOR. Hence, a total of 128 LUTs or 64 Spartan-3E slices are required, which is four times the size of a Ch implementation not protected by Boolean masking.

5.4 Addition modulo 2^{32}

All of the masks that have been introduced up to this point have been Boolean masks. However, the SHA-256 compression algorithm makes extensive use of consecutive additions modulo 2^{32}, denoted $\boxplus$, which are arithmetic functions and are non-linear with respect to Boolean masking. This presents the designer with a choice: (i) a new masked function $\boxplus'$ can be created, which uses Boolean masking; or (ii) a Boolean-to-Arithmetic conversion can be applied prior to the $\boxplus$ operations. The latter choice converts a variable masked with a Boolean mask to a variable masked with an arithmetic mask, meaning that subsequent additions are linear with respect to the arithmetic mask. Arithmetic-to-Boolean conversion is required before the results of the additions are fed back to the Boolean part of the function. In [7], Golić investigated this design choice, and concluded that choice (i) above is effective only if a small number of consecutive masked additions (e.g. one to three) is required. This is verified by our experiments on the FPGA (not detailed here). The masked adder produced in design

[1] In VHDL, this can be achieved by asserting the "keep_hierarchy" attribute within the masked AND gate's architecture.

(i) has large area and large latency, which greatly adds to the critical path in the circuit. On the other hand, design (ii) uses the conversion functions along with ordinary addition operations, both of which can take advantage of the underlying structure of the FPGA, leading to a much shorter critical path than in design (i). Our designs for the conversion functions are detailed below.

5.5 Boolean-to-Arithmetic Conversion

The circuits implementing the Boolean-to-Arithmetic and Arithmetic-to-Boolean conversion functions must themselves be secure against side-channel attacks. Several software-based algorithms have been proposed for performing these conversions [5, 8, 15]; however, these solutions are not suitable for efficient hardware implementation. Golić [7] developed hardware solutions based on the basic method of ripple-carry addition. Here, we present solutions tailored for FPGA implementation, based on the carry look-ahead addition method. We take advantage of the dedicated carry logic that is hard-wired into the FPGA, which allows carry bits to quickly propagate through columns of FPGA slices.

Henceforth, we will use single prime notation (x') to denote Boolean masking, and double prime notation (x'') to denote arithmetic masking (with respect to $\boxplus$). The goal is to securely convert a variable $x' = x \oplus r_x$ into a variable x'' such that $x'' = x \boxplus r_x$, without compromising the secret value x. Following the analysis in [7], we use a subscript j, $0 \leq j \leq 31$ to index the individual bits of x', x'' and r_x. The addition, with carry word c, can be expressed as:

$$x''_j = x_j \oplus r_{x,j} \oplus c_{j-1} \tag{26}$$

$$= x'_j \oplus c_{j-1} \tag{27}$$

where $c_{-1} = 0$, and c_{31} is not used. The carry bits are described by the recursive equation $c_j = (x_j \wedge r_{x,j}) \vee c_{j-1} \wedge (x_j \oplus r_{x,j})$. In order to suit a carry look-ahead implementation, the equation for the carry bits can be restated as:

$$\begin{aligned} c_j &= ((x'_j \oplus r_{x,j}) \wedge r_{x,j}) \vee (c_{j-1} \wedge (x'_j \oplus r_{x,j} \oplus r_{x,j})) \\ &= \overline{x'_j} \wedge r_{x,j} \vee x'_j \wedge c_{j-1} \end{aligned} \tag{28}$$

Clearly, equation (28) is suitable for implementation by a multiplexer in the FPGA's dedicated carry chain, with $r_{x,j}$ and c_{j-1} as data inputs, and x'_j as the control input. Therefore, the Boolean-to-Arithmetic conversion circuit should have similar area requirements and similar latency to an ordinary adder.

However, the above multiplexer-based design contravenes the secure computation condition, because the data input $r_{x,j}$ is not statistically independent of the control input x'_j. In theory, this dependence could be used by an attacker to launch a side-channel attack. In order to remove this dependence, we introduce a further Boolean masking bit q to mask the carry chain. The same bit q can be re-used for each multiplexer in the conversion circuit. The resulting scheme is described as follows:

$$c_j \oplus q = \overline{x'_j} \wedge (r_{x,j} \oplus q) \vee x'_j \wedge (c_{j-1} \oplus q) \tag{29}$$

$$x''_j = x'_j \oplus (c_{j-1} \oplus q) \oplus q \tag{30}$$

The carry look-ahead structure is maintained, which allows fast calculation of equation (29) on the FPGA. One extra LUT per bit is required by equation (30), to remove the mask from the masked carry bits.

5.6 Arithmetic-to-Boolean Conversion

The aim of an Arithmetic-to-Boolean conversion is to use x'' and r_x to derive the Boolean-masked variable x'. From equation (27), we have $x'_j = x''_j \oplus c_{j-1}$. In order to obtain a recursive expression for c_j in terms of x'', we substitute x'_j into equation (28), giving $c_j = \overline{(x''_j \oplus c_{j-1})} \wedge r_{x,j} \vee (x''_j \oplus c_{j-1}) \wedge c_{j-1}$. After some algebraic manipulation, the following can be derived:

$$c_j = (x''_j \oplus r_{x,j}) \wedge r_{x,j} \vee (\overline{x''_j} \oplus r_{x,j}) \wedge c_{j-1} \tag{31}$$

The conversion function is now in the carry look-ahead form that is required for fast calculation on the FPGA; with $r_{x,j}$ and c_{j-1} as the data inputs to the multiplexers, and $(\overline{x''_j} \oplus r_{x,j})$ as the control input. As above, we must now determine if the multiplexers comply with the secure computation condition. It appears that data input $r_{x,j}$ is statistically independent of the control input $(\overline{x''_j} \oplus r_{x,j})$, because x''_j incorporates randomness from bit c_{j-1} as well as from bit $r_{x,j}$ (equation (26)). However, when $j = 0$, c_{-1} is fixed at zero, and the control input becomes simply $\overline{x_0}$, i.e. one secret bit is unmasked.

Clearly, we must avoid computing $(\overline{x''_j} \oplus r_{x,j})$ when $j = 0$. Our solution is to remove one multiplexer from the carry chain, and to use an FPGA LUT to calculate c_0 directly. From equation (31), $c_0 = \overline{x''_0} \wedge r_{x,0}$, which could itself be the focus of a side-channel attack. Therefore, as in the case of the Boolean-to-Arithmetic conversion, we introduce a Boolean masking bit q, giving:

$$c_0 = (\overline{x''_0} \wedge r_{x,0}) \oplus q \tag{32}$$

Technically, this equation violates the secure computation condition, as the intermediate result $(\overline{x''_0} \wedge r_{x,0})$ is not independent of the secret bit x_0. However, if an FPGA LUT is used to calculate c_0, it can be shown that the LUT output is statistically independent of x_0, therefore the LUT does not leak information.

Finally, the other 31 masked values of c_j can be calculated using the fast carry chain, according to:

$$c_j \oplus q = (x''_j \oplus r_{x,j}) \wedge (r_{x,j} \oplus q) \vee (\overline{x''_j} \oplus r_{x,j}) \wedge (c_{j-1} \oplus q) \tag{33}$$

As in the case of Boolean-to-Arithmetic masking, additional LUTs are required to remove the masking bit q, via $x'_j = x''_j \oplus (c_{j-1} \oplus q) \oplus q$.

6 Masked FPGA Implementation

The above section detailed the proposed masking schemes for the SHA-256 compression function. Note that in order to remove the masks r_A–r_H at the end

of the 64th iteration of the compression function, it is necessary to compute mask update terms in parallel with the masked compression function. The complete masked core design contains: sixteen 32-bit registers; thirteen adders; seven Boolean-to-Arithmetic conversion blocks; two Arithmetic-to-Boolean conversion blocks; as well as circuits implementing the $\sum_0$, $\sum_1$, Maj' and Ch' functions.

On our Spartan-3E chip, the masked processor and interface circuitry utilise 1734 slices (37% of FPGA resources), and the design's critical path is 18.6 ns. Hence, although the area has almost doubled compared with the unprotected implementation, the speed has not been overly affected. The required random bits could be generated, for example, by a cryptographically secure pseudo-random number generator implemented on the FPGA, as described in [20]. For simplicity, in our case we pre-generated the required random bits, and stored them in BRAM on the FPGA. By repeating the experiments described in section 3, we verified that the data-dependence of the power consumption has been removed; therefore, the design is resistant to standard first-order DPA attacks.

We note that more sophisticated first-order DPA attacks are still possible, for example by considering the side-channel leakage due to glitches [13]. However, such attacks rely on the strong assumption that the attacker has very detailed knowledge of the design, such as a back-annotated netlist, from which an exact power model can be extracted.

7 Conclusions

It has been shown that implementations of the HMAC algorithm are susceptible to side-channel attacks. An explicit DPA attack strategy for HMAC-SHA-2 has been presented, and the attacks have been verified with actual FPGA-based experiments. A hardware-based masked core for SHA-2 hash functions has been designed, which counteracts first-order DPA attacks. The Boolean-to-Arithmetic and Arithmetic-to-Boolean conversion circuits, which are traditionally considered to be slow, have been optimised for implementation on FPGAs. This useful adaptation can be used to mask other algorithms that mix Boolean and arithmetic functions, such as IDEA or RC6. Future work will focus on securing the HMAC algorithm against other forms of side-channel attack, such as higher-order DPA and template attacks. Another avenue for further research is to investigate how throughput optimisation techniques can be applied to SHA-2 implementations, while maintaining the DPA attack countermeasures.

Acknowledgements

The authors would like to acknowledge the comments of the anonymous reviewers, as well as the reviewers of an earlier draft of this paper. This work was supported in part by the Embark Initiative, operated by the Irish Research Council for Science, Engineering and Technology (IRCSET).

References

1. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO'96, 16th Annual International Cryptology Conference*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.
2. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004, 6th International Workshop*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.
3. Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2002, 4th International Workshop*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.
4. Ricardo Chaves, Georgi Kuzmanov, Leonel Sousa, and Stamatis Vassiliadis. Improving SHA-2 hardware implementations. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems — CHES 2006, 8th International Workshop*, volume 4249 of *Lecture Notes in Computer Science*, pages 298–310. Springer, 2006.
5. Jean-Sébastien Coron and Alexei Tchoulkine. A new algorithm for switching from arithmetic to boolean masking. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2003, 5th International Workshop*, volume 2779 of *Lecture Notes in Computer Science*, pages 89–97. Springer, 2003.
6. Tim Dierks and Eric Rescorla. The Transport Layer Security (TLS) Protocol, Version 1.1. RFC 4346, Retrieved online, July 2007. `http://tools.ietf.org/html/rfc4346`, April 2006.
7. Jovan Dj. Golić. Techniques for random masking in hardware. *IEEE Transactions on Circuits and Systems — I*, 54(2):291–300, February 2007.
8. Louis Goubin. A sound method for switching between boolean and arithmetic masking. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001, Third International Workshop*, volume 2162 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2001.
9. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO'99, 19th Annual International Cryptology Conference*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
10. 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 Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004, 6th International Workshop*, volume 3156 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2004.
11. Arjen K. Lenstra. Further progress in hashing cryptanalysis (white paper). Retrieved online, July 2007. `http://cm.bell-labs.com/who/akl/hash.pdf`, February 2005.
12. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.
13. Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully attacking masked AES hardware implementations. In Josyula R. Rao and Berk Sunar,

editors, *Cryptographic Hardware and Embedded Systems — CHES 2005, 7th International Workshop*, volume 3659 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2005.
14. Vishwas Manral. Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH). RFC 4835, Retrieved online, July 2007. `http://tools.ietf.org/html/rfc4835`, April 2007.
15. Olaf Neiße and Jürgen Pulkus. Switching blindings with a view torwards IDEA. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004, 6th International Workshop*, volume 3156 of *Lecture Notes in Computer Science*, pages 230–239. Springer, 2004.
16. National Institute of Standards and Technology. FIPS PUB 180-2. Secure Hash Standard, August 2002.
17. National Institute of Standards and Technology. FIPS PUB 198. The Keyed-Hash Message Authentication Code (HMAC), March 2002.
18. Katsuyuki Okeya. Side channel attacks against HMACs based on block-cipher based hash functions. In Lynn Margaret Batten and Reihaneh Safavi-Naini, editors, *Information Security and Privacy, 11th Australasian Conference, ACISP 2006*, volume 4058 of *Lecture Notes in Computer Science*, pages 432–443. Springer, 2006.
19. Katsuyuki Okeya and Tetsu Iwata. Side channel attacks on message authentication codes. In Refik Molva, Gene Tsudik, and Dirk Westhoff, editors, *Security and Privacy in Ad-hoc and Sensor Networks, Second European Workshop, ESAS 2005, Revised Selected Papers*, volume 3813 of *Lecture Notes in Computer Science*, pages 205–217. Springer, 2005.
20. Dries Schellekens, Bart Preneel, and Ingrid Verbauwhede. FPGA vendor agnostic true random number generator. In *16th International Conference on Field Programmable Logic and Applications (FPL 2006)*, pages 139–144. IEEE, August 2006.
21. Rainer Steinwandt, Willi Geiselmann, and Thomas Beth. A theoretical DPA-based cryptanalysis of the NESSIE candidates FLASH and SFLASH. In *Information Security, 4th International Conference, ISC 2001*, volume 2200 of *Lecture Notes in Computer Science*, pages 280–293. Springer, 2001.
22. Michael Tunstall, Neil Hanley, Robert McEvoy, Claire Whelan, Colin C. Murphy, and William P. Marnane. Correlation power analysis of large word sizes. Submitted to *IET Irish Signals and Systems Conference (ISSC) 2007*. IEEE, 2007.
23. Xilinx. Spartan-3 Generation FPGA User Guide. Retrieved online, July 2007. `http://direct.xilinx.com/bvdocs/userguides/ug331.pdf`.