Unprotected and Masked Hardware Implementations of Spook v2

Charles Momin, Gaëtan Cassiers, and François-Xavier Standaert

Crypto Group, ICTEAM Institute, UCLouvain, Louvain-la-Neuve, Belgium

Abstract. We describe FPGA implementations of the Spook candidate to the NIST lightweight cryptography competition in two flavors. First, unprotected implementations that exhibit the excellent throughput and energy consumption for the area target specified by the NIST benchmarking initiative. Second, protected implementations leveraging the leveled implementation concept that the Spook design enables and confirming the significant performance gains that it enables.

1 Introduction

Spook is Authenticated Encryption with Additionnal Data (AEAD) scheme. It is a Round 2 candidate to the NIST LightWeight Cryptography (LWC) competition.[1] Its primary design goals are resistance against side-channel attacks (SCAs) and energy efficient implementation. Spook is build on the TETSponge AEAD mode, that is based on two primitives: a Tweakable Block Cipher (TBC) used in the key generation and tag generation parts of the mode, and a permutation used in the the sponge part of the mode [8]. The TBC of Spook is Clyde-128 and (in the primary variant of the mode) its permutation is Shadow-512. Both algorithms are named after their block size. Recently, an update of Spook (Spook v2) has been introduced [3], in order to increase the security margins of Shadow-512 while also improving the performances of some of its components.

Since the NIST LWC competition is aimed at standardizing "lightweight cryptographic algorithms that are suitable for use in constrained environments", evaluating their implementation results is of primary importance for the comparison of different candidates. Relevant constrained environments include hardware platforms such as FPGAs and ASICs. Therefore, in order to make the comparison of hardware cipher implementations easier and more fair, a common hardware API (next denoted as the LWC HW API) was proposed by the Athena Project benchmark initiative, which we use next [9].

Since the embedded platforms are one of the main targets for the NIST LWC competition, side-channel attacks (SCA) are important to consider. The Spook cipher is designed with this concern in mind, and it is expected that it can be implemented with excellent protection against SCAs and limited overheads. For this purpose, one of the main countermeasures against SCAs is masking. It is

[1] `https://csrc.nist.gov/projects/lightweight-cryptography`.

an algorithmic protection that consists in replacing every intermediate value in a computation by a set of so-called shares that are individually independent of the secret intermediate value [7]. Although the design of efficient masking schemes has been extensively studied, implementing a cryptographic primitive in a masked fashion still incurs significant overheads. The TETSponge mode used in Spook aims at enabling so-called leveled implementations, which minimize these overheads by ensuring strong security guarantees against leakage with only two calls to a strongly protected (e.g., masked) TBC while only requiring weak protections (or in parallel hardware implementations, no protection at all) for the other permutation calls used to process the message.

In this paper, we present two hardware implementations of Spook: an unprotected one and a leveled, protected one (where Clyde-128 is masked). Both implementations are compliant with the LWC HW API and have been designed in a general architecture where API handling features are common for the two implementations, and only the block implementing core primitives differs. In addition to performance improvements for the unprotected core compared to the preliminary results reported in [10] and [3], we also present the first results of side-channel protected implementations of the full Spook algorithm.

The paper is organized in four parts. First, we describe the general architecture. Next, details about both the unprotected and the protected cores are given. Finally, performance metrics for both implementations are presented.

2 General framework

In this section, we describe at a high level the main blocks of the implementations and how they interface. The implementations are built around a crypto core block which implements the cryptographic functionality and is optimized to be as small and efficient as possible, at the expense of an increase of complexity in its interfaces: it has many control signals, and the input data must be provided in fixed-sized, already padded blocks. The interface of the crypto core is however identical for both normal and masked implementations. The other blocks adapt the crypto core to the more standard interface of the LWC HW API [9].

We next briefly recall the LWC HW API, then describe the various blocks that interact to implement it: the crypto core, segment manager, main FSM, encoder, decoder and SDI handler, as shown in Figure 1.

2.1 LWC HW API

The LWC HW API is based around three interfaces: the public data input (PDI), the secret data input (SDI), and the data output (DO). An additional interface (RDI) dedicated to provides fresh randomness is considered in the case of protected implementations. Each interface features a bus of a given width (32 bits in our implementation), a valid signal and a ready signal. The interface is synchronous and unidirectional, with the bus and valid signals going in the direction of the interface and the ready signal going in the opposite direction. A transfer

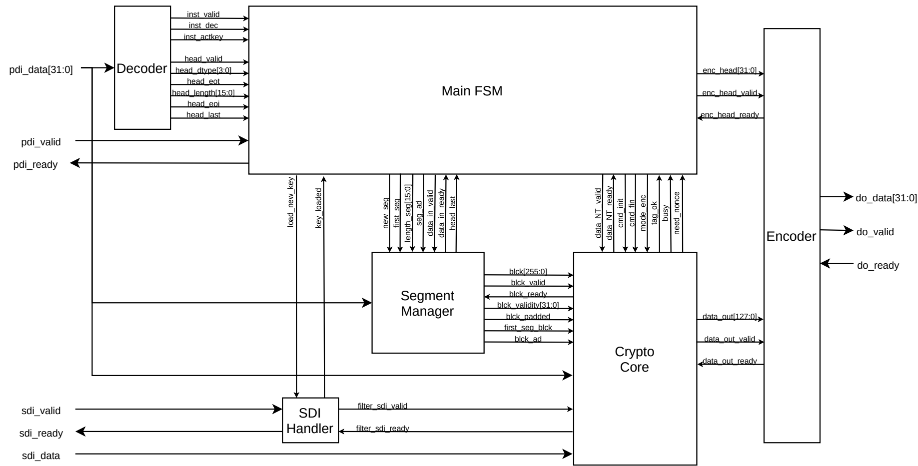

Fig. 1. General framework.

occurs on the interface if, at a given clock cycle, both valid and ready signals are high. The data transferred is the one present on the bus at that clock cycle.

On each interface, two kinds of data are transferred: instructions and segments. An instruction consists in a single 32-bit transfer while a segment starts with a transfer containing the segment header, followed by a number of payload transfers (the number of payload transfers is contained in the segment header). Three main instructions are considered: ACTKEY (used to start the loading of a new key through the SDI interface), ENC and DEC (used to start respectively an encryption or a decryption process). Two additional status instructions indicate the successful completion of an instruction: SUCCESS and FAILURE. Finally, only the segments types allowing the independent transfer of the following data are considered: the key (only on the SDI interface), the nonce, the associated data, the plaintext, the ciphertext and the tag. No merged data segment (e.g., Npub || AD) are supported.

The typical operation of the LWC HW API is as follows: first, the key is loaded by sending an activate key instruction on the PDI, followed by a key load instruction and a key segment on the SDI. Then, an encrypt (resp., decrypt) instruction is sent on the PDI, followed by a Nonce segment, one empty segment (i.e., of length 0) or any number of AD segments, with one empty or any number of plaintext (resp., ciphertext) segments. Finally, for decryption operation, a tag segment is sent. For each plaintext (resp., ciphertext) segment on the PDI, a ciphertext (resp., plaintext) segment is produced on the DO. An additional tag segment followed by a SUCCESS status are produced on the DO at the end of an encryption operation. Decryption operation additionally outputs a status instruction on the DO indicating if the decryption succeeded or failed.

2.2 Crypto core

The crypto core implements the cryptographic functionality and comes in two flavours: the unprotected core (Section 3) and the protected core (Section 4) which contains a masked implementation of the Clyde-128 TBC.

The main input of the crypto core is a bus carrying blocks of 256 bits of AD, plaintext or ciphertext (matching the rate of the TETSponge mode). Those blocks must be padded, and metadata about which bytes are padding is required. A 32-bit SDI input bus carries the key, while another 32-bit bus inputs the nonce and the tag. The only output is a 128-bit bus, which corresponds to half a block (matching the crypto core internal pipeline width) or to the size of the tag.

The control signal of the core, in addition to the validity/readiness of each bus, are initialization and finalization commands, and encryption/decryption mode selection. Additionally, there is a tag_ok signal that is the output of the tag verification circuit, a busy signal to indicate readiness of the core and a need_nonce signal to report to the main FSM that the value of the nonce is still required and cannot be overwritten.

The wide input bus of the core aims at maximizing its flexibility, leaving the integrator to use the buffering strategy that is most suited for each use-case. Since implementing the padding feature on such a wide bus would be expensive, it has been decided not to implement it in the crypto core, which allows for more optimized implementations.

2.3 Segment Manager

The segment manager is set up (by the main FSM) with the length (in bytes) of a segment, then takes inputs from a 32-bit bus and concatenates them in 256-bit blocks (padding if necessary). The segment manager is built around a 256-bit shift register (the block builder) and padding logic operating over 32 bits.

Its block output is fed to the crypto core, along with metadata: which bytes of the block are padding, whether the block is AD, whether the block is the first of its type (that is, first block of AD/plaintext/ciphertext).

The control interface to the main FSM is made of the initilization new_seg signal, the length of the segment and whether the segment is the first of its type. The segment manager signals the last segment transfer with the feed_last signal.

2.4 Other supporting blocks

The main FSM controls the crypto core, the Segment manager and the SDI handler. It also sends output segment headers to the encoder. *The decoder* is a combinational block that decodes instructions and segment headers. *The encoder* builds the output from segment headers and result instructions provided by the main FSM and segment data from the crypto core. It contains a full 256-bit buffer for the segment data (in order to avoid stalling the crypto core while the output is transferred) and a 32-bit instruction buffer (to avoid stalling the main FSM). *The SDI handler* filters instructions and segment headers from the SDI input, and controls the transfer of the key to the crypto core.

2.5 Timing and interfaces

The data interfaces between the blocks follow the data/valid/ready principle of the LWC HW API. However, for performance reason (reducing critical path), the data signal is one cycle late with respect to the valid/ready signal on the segment manager to cryto core and crypto core to encoder interfaces.

2.6 Compliance with the LWC HW API

Our implementation follows the minimum compliance criteria described in [9]. The interface bus width is limited to 32 bits. The size of the AD/PT data is not limited by the implementation (provided that these are split in one or multiple segments having a maximum size of $2^{16} - 1$ bytes of payload each, as mandated by the LWC HW API). After the encrypt (resp., decrypt) instruction and once the key has been loaded, the core expect the following segment order at the PDI interface during an encryption process: public nonce, associated data and plaintext (resp., ciphertext). The data segments produced at the DO interface are the ciphertext (resp., plaintext) followed by the tag (resp., no other segment).

The only non-compliance to be noted with the LWC HW API is the way the shared key data is sent to the core for the protected implementation. The LWC HW API transfers all the shares of each key word sequentially, which could lead to security order reduction: if leakage on the SDI bus depends on the XOR between sequential values (a.k.a. transition leakage), then the security order may be halved [1]. Instead, our implementation transfers one share of all words at a time, and process all shares sequentially. The transition occurs then between shares of of different words, which mitigates their impact.

3 Unprotected core

In this section, we describe the unprotected Spook core. Its architecture is oriented towards the area requirements specified for the NIST benchmarking effort (`https://cryptography.gmu.edu/athena/LWC/LWC_Suggested_FPGA_Design_Goals.pdf`) while optimizing throughput and energy consumption.

3.1 Overview

As represented in Figure 2, the unprotected core is composed of two main parts: the primitive core unit (Figure 3) contains most of the core's datapath; the core FSM drives the datapath according to the commands issued by the Main FSM. Other parts are two 128-bit registers to store the nonce or tag (N/T holder) and the key (key holder), respectively. A 128-bit MUX selects the output of the core as either the plaintext/ciphertext (aka digestion result) or the tag.

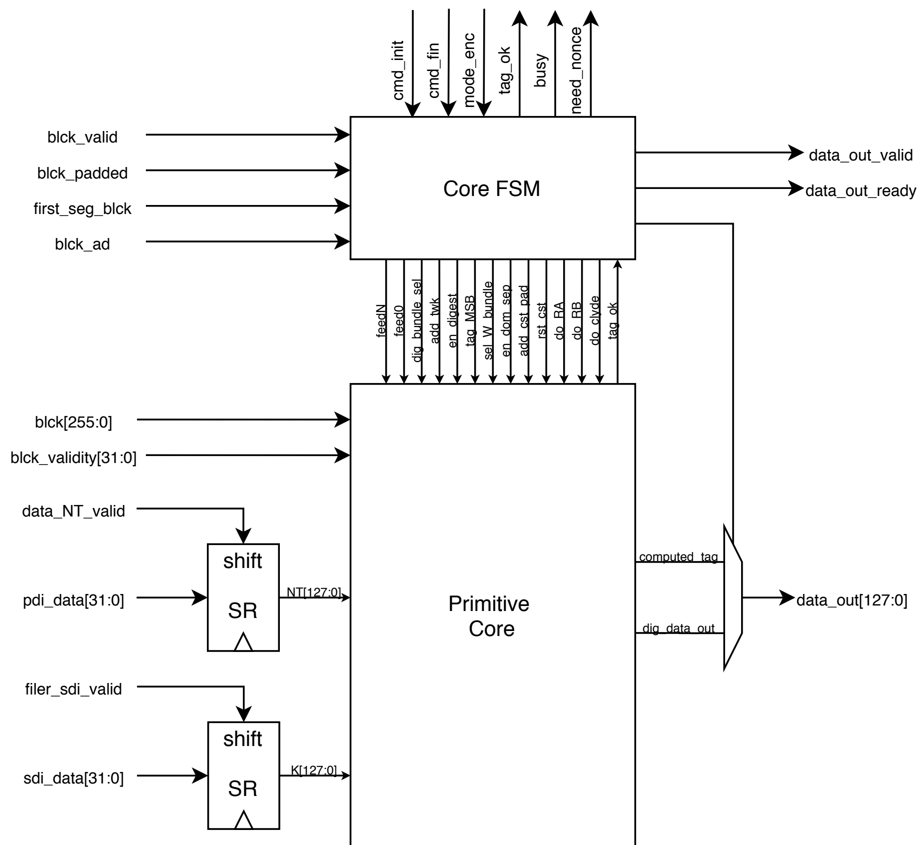

Fig. 2. Unprotected crypto core.

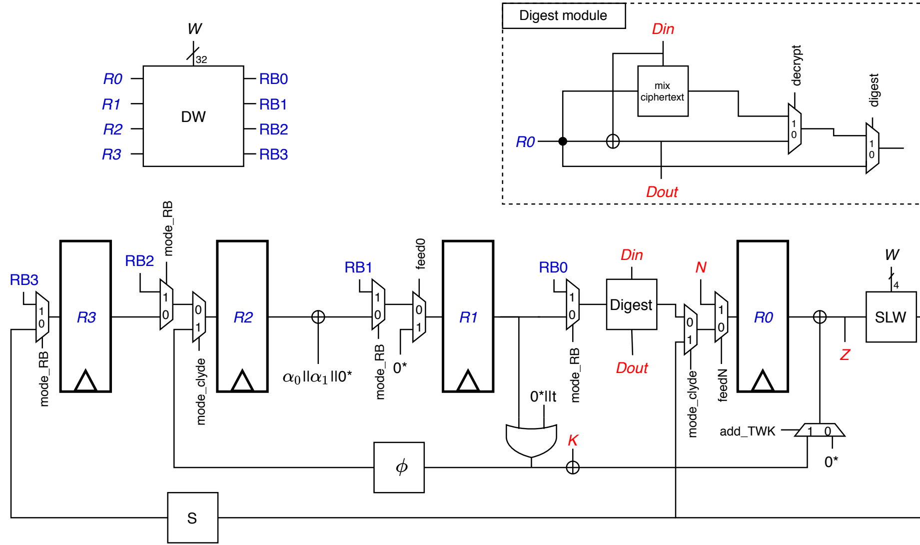

Fig. 3. Unprotected crypto core: primitive core.

3.2 Primitive core

The main functionality of the primitive core is the implementation of the Clyde-128 (tweakable block cipher) and Shadow-512 (permutation) cryptographic primitives. The primitive core architecture is introduced in [3], to which we refer for the description of the implemented algorithms and for a high-level overview. For completeness, we recall and expand the description of its architecture.

The architecture operates on 128 bits, hence it requires 4 cycles to execute a round of Shadow-512 and 1 cycle for a round of Clyde-128. Furthermore, thanks to the similarities between Clyde and Shadow (same S-box, L-box, etc.), the ability to compute Clyde-128 can be added on top of the implementation of Shadow-512 with little additional logic.

On Figure 3, each bus is 128-bit wide unless indicated otherwise. The IOs (in red) are the nonce, key, tag, bytes to digest (denoted as `Din`, that are AD padded plaintext or ciphertext) and the digested bytes (denoted as `Dout`, that are plaintext or ciphertext). The symbols S, L, D and W in logic blocks correspond respectively to the S-box, L-box, D-box and constant addition.

The `mix ciphertext` module is only used during a decryption process to mix the partial input ciphertext block with the state of the sponge before it is used during the following execution of Shadow-512. The signals α_0 and α_1 control the addition of the domain separation bits.

To compute Clyde-128, the initial plaintext and tweak are respectively stored in the registers `R0` and `R1`. The control signal `mode_RB` is unset (i.e., equals 0) while `mode_clyde` is set. In this way, the state of Clyde-128 cycles through the `R0` register and the SLW logic (and the tweakey addition) at the rate of one round

per clock cycle. Therefore, the full Clyde-128 computation takes 12 cycles. The tweak flows through registers R1, R2 and through the ϕ logic, producing a valid updated tweak every two clock cycles.

For Shadow-512, the four 128-bit bundles $b_0, \ldots, b_3$ are stored in the registers R0 to R3. Those registers act as a large circular shift register with two shifting modes. Shadow-512 uses two rounds that differ in their linear layer. For Round A (and the S-box of round B), the data cycles from R3 to R0, then through the SLW unit and the S unit back to R3, computing a full round in four cycles (all mux controls are unset). For Round B (except its S-box), data is cycling inside the same Ri register: the signal mode_RB is set, forwarding the data to the DW unit (that updates a part of it input and shifts the other part) for 4 cycles.

When starting the Spook operation, the N and 0^* values are loaded in R0 and R1 respectively during the first clock cycle. Then, in 12 cycles, the first call of Clyde-128 is computed, producing the fresh seed B.

Next, during the first round of the first execution of Shadow-512, the initial state (i.e., $0^*||N||0^*||B$) is loaded sequentially per bundle (except for B), using again the control signals feed0 and feedN. Shadow is then executed (in 48 cycles per execution), and the digest unit is used a the beginning of each execution when AD/P/C needs to be fed. Finally, the Clyde tag computation takes 12 cycles, with the signal t set in order to ensure that the MSB of the tweak is high.

Two LFSRs are used in order to generate the constants of the primitives: a 4-bit LFSR for Clyde and a 32-bit LFSR for Shadow.

4 Protected core

In this section, we describe the protected Spook core architecture. The latter aims at providing high security against leakage by only masking the Clyde-128 TBC. Precisely, such an implementation offers strong integrity guarantees (i.e., ciphertext integrity with misuse-resistance and leakage in encryption and decryption) and the best confidentiality guarantees that can be reached with a one-pass design (i.e., CCA security with misuse-resilience and leakage in encryption only). See [4] for a discussion. As for performance, such a leveled implementation aims maintaining the throughput and energy efficiency of the unprotected core at the cost of a larger (but manageable) circuit.

4.1 Overview

As explained in [3], a protected Spook implementation can leverage a leveled architecture. This means that, thanks to the TETSponge mode, the side-channel security (and therefore the area cost, speed and energy consumption) of the Clyde-128 and Shadow-512 implementations can significantly differ. The first one protects a long-term key and has to be strongly protected against side-channel attacks (i.e., ensure DPA security) – we will use masking for this purpose. The second one protects ephemeral secrets and requires weaker (and cheaper) protections (i.e., ensure SPA security) – we will use an unprotected parallel

implementations for this purpose. In order to ensure the integrity guarantees, one can choose between implementing a masked tag verification or to exploit the (unprotected) inverse-based tag verification from [5]. We used the slightly cheaper and concetually simpler inverse-based solution.

The protected core contains two independent sub-cores: the Shadow core and MSKClyde core, as depicted in Figure 4. The first one computes the Shadow512 primitive and and does not use any special countermeasures. The other computes both the direct and the inverse operations of the Clyde128 primitive and is masked. The Shadow core follows the same architecture as the primitive core described in Section 3. The main difference is that the logic related to the computation of Clyde-128 (i.e., The Φ logic, the tweakey addition, the 4-bits constant addition and the bypass multiplexer) are removed. The MSKClyde is deeply changed: its architecture will be detailed in Section 4.2.

While the N/T data holder is the same as for the unprotected case, the key holder is modified in order to manage shared key values: in addition to increasing its size, a refresh mechanism is added, in order to refresh the key after each Clyde-128 evaluation. The refresh algorithm is as follows: let d be the number of shares, let $x_0, \ldots, x_{d-1}$ be the shares and $r_0, \ldots, r_{d-1}$ be random bits (taken from a 128-bit LFSR PRNG), the refreshed shares are $x_i \leftarrow x_i \oplus r_i \oplus r_{(i+1 \mod d)}$, (which we assume sufficient in practice, see [2], Theorem 4) A Random Data Input (RDI) interface is added and is handled by a RDI handler that, when activated, reseeds the PRNGs of the key holder and of the MSKClyde.

4.2 Architecture of the masked Clyde module

The MSKClyde core relies on the HPC2 glitch-resistant masking scheme proposed in [6], which provides state-of-the-art guarantees of composability in the presence of physical defaults like glitches. As depicted in Figure 5, it takes as input an unshared tweak (i.e., clyde_tweak), an unshared plaintext/ciphertext (i.e., clyde_din), a shared key (i.e., key_sharing) and outputs a shared ciphertext/plaintext (i.e., dout) that is then recombined (i.e., clyde_dout). Similarly to most masking schemes, the HPC2 scheme requires fresh randomness to perform the non-linear operations in a secure manner. Randomness is provided by the rnd0 and rnd1 busses, and is generated by dedicated PRNGs (i.e., PRNG0 and PRNG1). Each PRNG is implemented with parallel 128-bits LFSRs, each producing 32 fresh random bits per clock cycle. We focused on the case where these LFSRs allow to generate all the required randomness (i.e., $d*(d-1)$ bits per bus and per Sbox) in 1 clock cycle. As a result, the exact amount of parallel LFSRs depends on the number of shares chosen for the masking scheme.

A wrapper (the Stalling Unit) is used to ensure that the randomness is valid when required (i.e., when the signal need_rnd* is asserted) and that it does not enter the core earlier. In practice, the core asserts the signal pre_need_rndi 1 cycle before fresh randomness is required on the bus rndi. This has the effect of starting the generation of fresh randonmness by the PRNGi. If the randomness is not valid at the next clock cycle (i.e., the signal rndi_valid is not asserted), the MSKClyde core is stalled as long as it is the case.

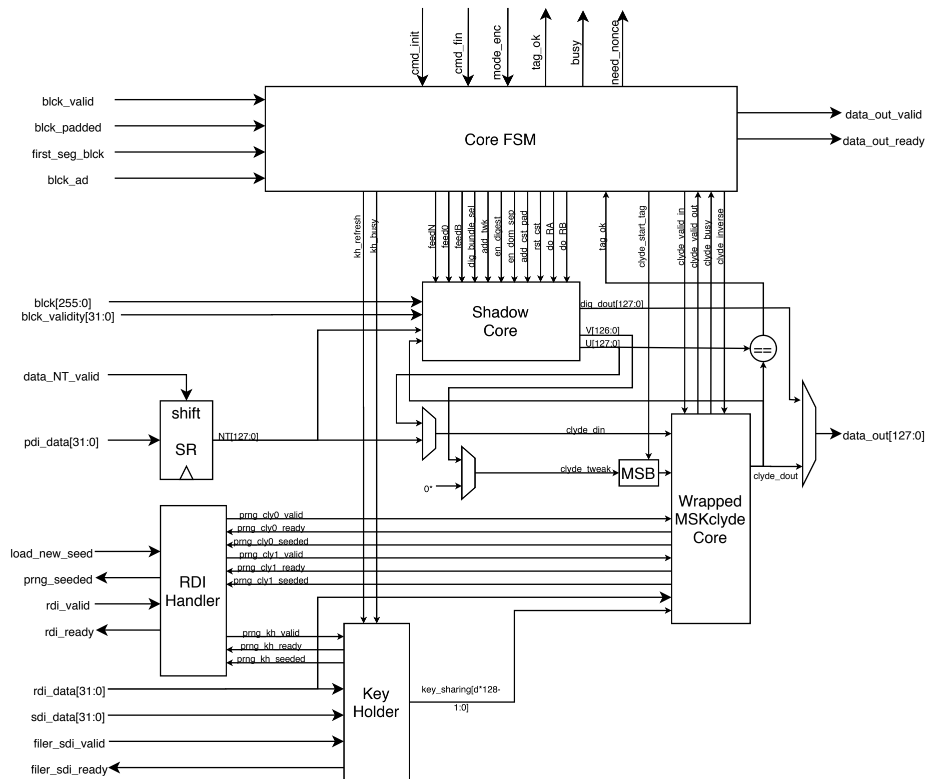

Fig. 4. Protected crypto core.

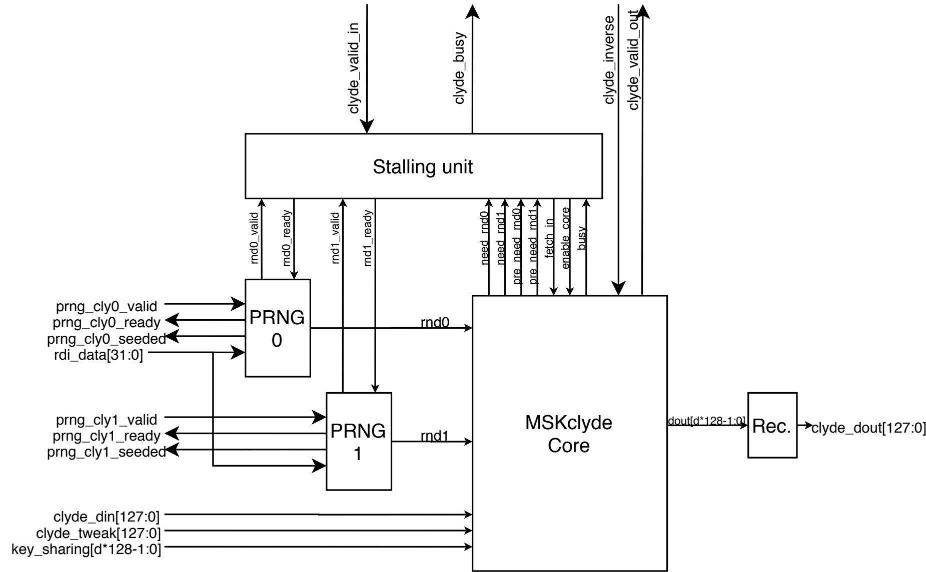

Fig. 5. Wrapped protected Clyde core.

As shown in Figure 6, the MSKClyde core operates over a $d * 128$-bit long shared representation of the state (i.e., state_sharing), where d is the number of shares. To make this possible considering the unshared inputs, the processed data first goes through a cst_msk unit that trivially shares its input. When a new execution starts, the signal new_run is asserted during one clock cycle. This has the effect of fetching the inputs in the core and resetting the submodules. This step is followed by the first constant and tweakey additions performed by the Add_W_TWK core, whose input is generated by the W unit and the δ unit, providing respectively the values of the 4-bit constant (i.e., W) and the 128-bit δ used to compute the tweakey. Both the W unit and the δ unit are similar to the 4-bits LFSR and the Φ unit of the unprotected core, except that an additional register holds the evolving values of δ. The FSM Clyde core controls the update of the later by asserting the signals upd_W and upd_delta during one clock cycle. The signals add_W and add_TWK enable the addition of the corresponding value inside the Add_W_TWK core. The shared value of the key is fed by the Key Holder and does not change during the Clyde-128 computation.

The Clyde-128 rounds (composed of an Sbox layer, an L-box layer and a constant addition) are computed by alternating between the Sbox layer logic and the Lbox layer logic. The Clyde FSM core drives the datapath according to the layer under computation and enables the addition of the constant and of the tweakey when required. Two architectural parameters can be changed: the number of parallel masked 4-bit S-boxes implemented in the S-box layer logic (denoted by N_{SB}) and the number of parallel masked 64-bit L-boxes in the

Lbox layer (denoted by N_{LS}). By using multiple S-boxes/L-boxes in parallel and combining them with a shift register strategy, the area versus latency trade-off of the Clyde-128 core can be adjusted. In particular, the latency of the L-box layer is $2/N_{LS}$ with N_{LS} being equal either to 1 or 2. For the S-box layer, the latency is $3+32/N_{SB}$ where N_{SB} can take any of the following values: 1, 2, 4, 8, 16 and 32. The additional three cycles are due to the latency of a single masked S-box that comes from masking scheme requirements [6].

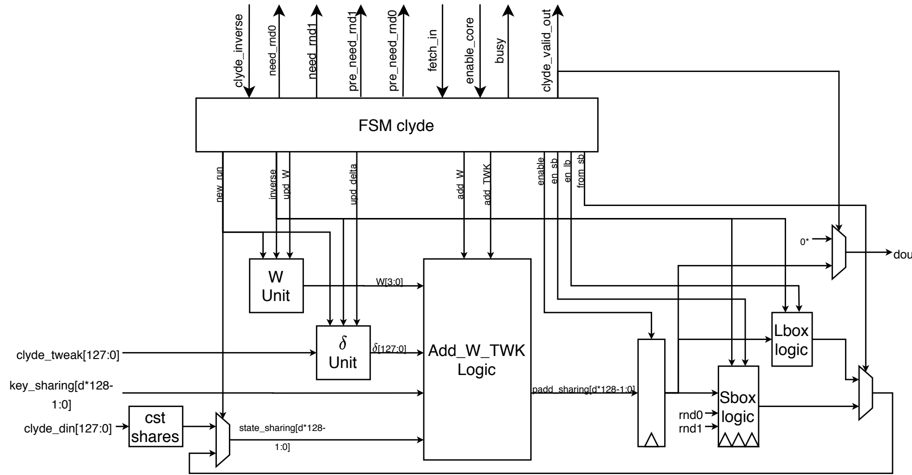

Fig. 6. Protected Clyde-128 core.

Each block developed above can compute its inverse operation, making straightforward the implementation of the inverse Clyde-128 primitive. Except for the masked S-boxes, the inverse functionality of each unit is implemented using an independent and parallel dedicated logic block. Both the direct and the inverse functionality logic blocks are fed with the input data and their corresponding output are muxed. The appropriate value is chosen depending on the value of the signal inverse.

In order to reduce the logical cost, the implementation of the inverse S-box reuses the already existing logic for the implementation of the direct one. The S-box being nearly involutive, its inverse is obtained by applying a simple linear layer at the inputs and outputs of the already existing core, as depicted in Figure 7. This costs 5 XOR gates and 2 multiplexers while allowing to avoid the costly logic of the masked AND gates.

5 Results

In this section, we present the results obtained for FPGA and ASIC implementations. The numbers we provided for both targets rely on the architectures

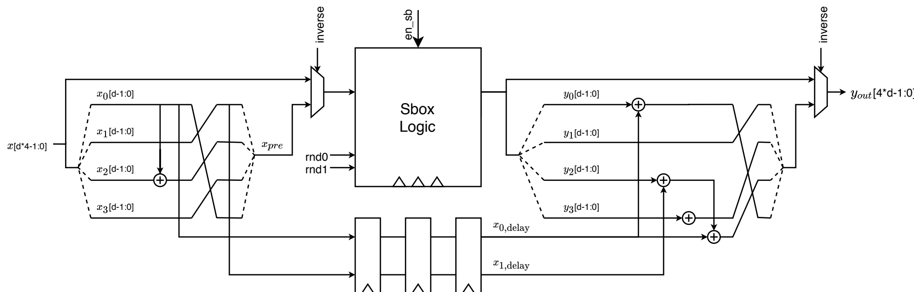

Fig. 7. Protected S-box supporting direct and inverse operation.

described above. In particular, the protected implementation was synthetized adopting $N_{SB} = 8$ and $N_{LB} = 1$ as degree of parallelization.

Note that an additional pipeline stage is considered between the core FSM and the datapath of each crypto core. Combined with the possibility to replicate some registers of the latter, is ease the routing of some sensitive control signal driving multiplexers involved in the critical path.

The numbers provided for the FPGA target are based on an Artix7 FPGA (xc7a50tcsg325-3) implementation performed with the Xilinx Vivado v2019.2.1 toolset. These results are shown in Table 1 and are given post place-and-route (i.e., post-implementation). The table includes standard FPGA metrics, namely the amount of slice registers and look-up tables required, the maximal clock frequency, the latency and the corresponding throughpout (for long messages) as well as the TPA (throuput to area ratio).

The maximum throughput is improved by 112 Mbps (11.7%) compared to the results reported in [3]. This is achived with a slight resources increase (45 additional registers (3%) needed for the pipelining of control signals). The protected core is larger than the unprotected one, but it achieves the same throughput.

Instance	Regs	LUTs	Freq [Mhz]	Latency [cycles]	Throughput [Mbps]
Unprotected	1527	2067	206.8	48	1102.9
Protected ($d = 2$)	5499	6340	200	48	1066.6
Protected ($d = 3$)	6209	9111	200	48	1066.6
Protected ($d = 4$)	8367	11555	200	48	1066.6

Table 1. Artix-7 implementation results (xc7a50tcsg325-3)

The numbers provided for the ASIC target are based on a 65nm technology implementation performed with the TSMC-N65LP (low-power) design kit. The tools are Cadence Genus v18.10-p003_1 for the synthesis and Cadence Innovus

v18.10-p002_1 for the place-and-route flow. The implementation relies on the use of a clock-gating strategy in order to reduce the dynamic power when sub-blocks are in idle state. The results are shown in Table 2 and are given post synthesis. The table includes standard ASIC metrics, namely the area, the maximal clock frequency, the estimated power consumption, the throughput (for long messages) and the energy per bit. The latency is the same as for the FPGA implementation.

The results for the unprotected core exhibit a throughput improvement of 975 Mpbs (45%) compared to the results reported in [3]. This come at the significant area cost of 6.93 kGE (38%). Again, the protected core is larger than the unprotected one, but it achieves the same throughput.

Instance	Area [kGE]	Frequency [Mhz]	Power [mW]	Throughput [Mbps]	Energy [nJ/bit]
Unprotected	25.13	588	7	3124	2.24
Protected ($d = 2$)	64.6	588	12.46	3124	3.99
Protected ($d = 3$)	88.8	588	17.611	3124	5.64
Protected ($d = 4$)	116.03	588	23.44	3124	7.5

Table 2. ASIC TSMC-N65 implementation results (post synthesis)

For short messages, the throughput is lower than the one shown in Table 1 and Table 2 due to the latency inherent to the initial/final executions of Clyde-128. This takes 24 cycles for the unprotected core and $12 * (2/N_{LS} + 32/N_{SB} + 3)$ cycles for the protected core. In Figure 8, we illustrate the throughput of the implementations for various message lengths. It shows that the overhead cost of initialization/finalization is small when the message is larger than 1 kB. Furthermore, the number of implemented S-boxes in the protected Clyde-128 implementation can be reduced with limited impact on the latency.

Acknowledgements Gaetan Cassiers is a PhD student and François-Xavier Standaert is a Senior Research Associate of the Belgian Fund for Scientific Research (FNRS-F.R.S.). This work has been funded in parts by the ERC project SWORD and the Win2Wal project PIRATE.

References

1. Balasch, J., Gierlichs, B., Grosso, V., Reparaz, O., Standaert, F.: On the cost of lazy engineering for masked software implementations. In: CARDIS. Lecture Notes in Computer Science, vol. 8968, pp. 64–81. Springer (2014)
2. Barthe, G., Dupressoir, F., Faust, S., Grégoire, B., Standaert, F., Strub, P.: Parallel implementations of masking schemes and the bounded moment leakage model. In: EUROCRYPT (1). Lecture Notes in Computer Science, vol. 10210, pp. 535–566 (2017)

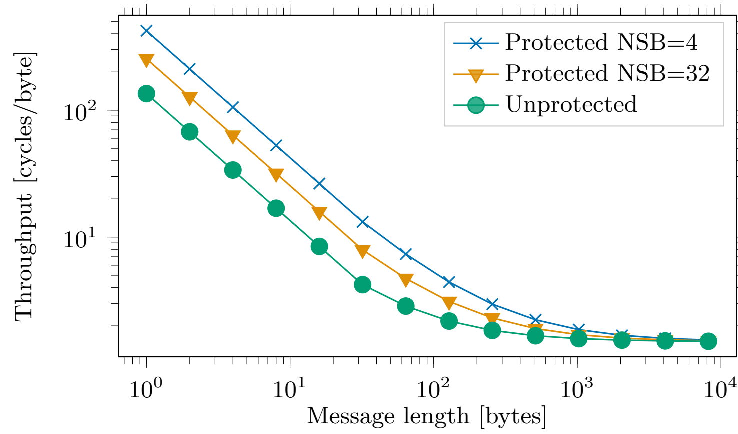

Fig. 8. Throughput of Spook encryption for protected and unprotected implementations with no AD. For the protected implementation, there is one L-box implemented, and the number of S-boxes is either 4 or 16.

3. Bellizia, D., Berti, F., Bronchain, O., Cassiers, G., Duval, S., Guo, C., Leander, G., Leurent, G., Levi, I., Momin, C., Pereira, O., Peters, T., Standaert, F., Udvarhelyi, B., Wiemer, F.: Spook: Sponge-based leakage-resistant authenticated encryption with a masked tweakable block cipher. IACR Trans. Symmetric Cryptol. **2020**(S1), 295–349 (2020)
4. Bellizia, D., Bronchain, O., Cassiers, G., Grosso, V., Guo, C., Momin, C., Pereira, O., Peters, T., Standaert, F.: Mode-level vs. implementation-level physical security in symmetric cryptography - A practical guide through the leakage-resistance jungle. In: CRYPTO (1). Lecture Notes in Computer Science, vol. 12170, pp. 369–400. Springer (2020)
5. Berti, F., Pereira, O., Peters, T., Standaert, F.: On leakage-resilient authenticated encryption with decryption leakages. IACR Trans. Symmetric Cryptol. **2017**(3), 271–293 (2017)
6. Cassiers, G., Grégoire, B., Levi, I., Standaert, F.: Hardware private circuits: From trivial composition to full verification. IEEE Transactions on Computers (2020 (to appear))
7. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: CRYPTO. Lecture Notes in Computer Science, vol. 1666, pp. 398–412. Springer (1999)
8. Guo, C., Pereira, O., Peters, T., Standaert, F.: Towards low-energy leakage-resistant authenticated encryption from the duplex sponge construction. IACR Trans. Symmetric Cryptol. **2020**(1), 6–42 (2020)
9. Kaps, J.P., Diehl, W., Tempelmeier, M., Homsirikamol, E., Gaj, K.: Hardware api for lightweight cryptography `https://cryptography.gmu.edu/athena/LWC/LWC_HW_API.pdf`
10. Rezvani, B., Diehl, W.: Hardware implementations of NIST lightweight cryptographic candidates: A first look `https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2019/documents/papers/hardware-implementations-of-nist-lwc-candidates-lwc2019.pdf`