A Lightweight Implementation of Saber Resistant Against Side-Channel Attacks

Abubakr Abdulgadir, Kamyar Mohajerani, Viet Ba Dang, Jens-Peter Kaps, and Kris Gaj

Cryptographic Engineering Research Group,
George Mason University
Fairfax, VA, U.S.A.
{aabdulga,mmohajer,vdang6,jkaps,kgaj}@gmu.edu

Abstract. The field of post-quantum cryptography aims to develop and analyze algorithms that can withstand classical and quantum cryptanalysis. The NIST PQC standardization process, now in its third round, specifies ease of protection against side-channel analysis as an important selection criterion. In this work, we develop and validate a masked hardware implementation of Saber key encapsulation mechanism, a third-round NIST PQC finalist. We first design a baseline lightweight hardware architecture of Saber and then apply side-channel countermeasures. Our protected hardware implementation is significantly faster than previously reported protected software and software/hardware co-design implementations. Additionally, applying side-channel countermeasures to our baseline design incurs approximately 2.9× and 1.4× penalty in terms of the number of LUTs and latency, respectively, in modern FPGAs.

Keywords: Post-Quantum Cryptography · lattice-based · Key Encapsulation Mechanism · hardware · FPGA · Side-Channel Analysis

1 Introduction

The accelerating development of post-quantum computing threatens the security of our current public-key infrastructure, based on traditional public-key cryptosystems, such as RSA and Elliptic Curve Cryptography (ECC). This threat motivates Post-Quantum Cryptography (PQC) research and development, aiming to produce and analyze algorithms that can withstand quantum and classical attacks and, at the same time, run on traditional computing platforms. The NIST PQC standardization process, currently in its third round, aims to coordinate the development and analysis of PQC algorithms to eventually select a few of them as new American Federal Information Processing Standards (FIPS).

Side-channel analysis (SCA), including Differential Power Analysis (DPA) [12], is a significant threat to the successful deployment of cryptographic solutions. Lightweight applications with limited or no physical security are even more susceptible to such attacks since adversaries can easily collect side-channel information. Consequently, the NIST PQC standardization process specifies ease of protection against side-channel attacks as a desirable feature of candidates. Among

the most urgent tasks in the evaluation process is developing SCA-resistant implementations of third-round finalists and assessing their comparative cost of protection against SCA. All the range of target platforms from pure software to full hardware and hybrid platforms need consideration since leakage patterns differ from one platform to another. For example, architectural leakage stemming from processor architecture can affect the software, while glitches, dependent on basic combinational and sequential circuit building blocks, affect hardware implementations.

NIST has selected Saber, a lattice-based key encapsulation mechanism (KEM), as a third-round finalist in July 2020. Previous works on applying SCA countermeasures to Saber concentrated on software [4] and software/hardware co-design [8].

In this work, we develop and evaluate SCA-resistant full hardware implementations of Saber. Our hardware design is significantly faster compared to previously reported SW, and SW/HW masked implementations of Saber. Additionally, our masked design uses approximately 2.9× more lookup tables (LUTs) while incurring 1.4× performance penalty compared to the unprotected baseline design when implemented in Xilinx Artix-7 FPGAs. Our results show the possibility of producing efficient masked hardware implementations of Saber that are significantly faster than SW and SW/HW designs. The source code of our implementation is publicly available at `https://github.com/GMUCERG/SABER-SCA`.

2 Previous Work

PQC algorithm side-channel resistance is an active research field with several open problems. Developing efficient countermeasures suitable for PQC algorithms and assessing the comparative cost of protection are critical for a fair comparison of NIST PQC third-round candidates. The community has made progress towards these goals, but many open questions remain.

In [16], Reparaz et al. proposed a masked implementation for ring-Learning-With-Errors (ring-LWE). The main idea is to split the secret polynomial $\mathbf{s}$ into two shares $\mathbf{s}_0$ and $\mathbf{s}_1$ such that $\mathbf{s} = \mathbf{s}_0 + \mathbf{s}_1$. Multiplying the shared version of $\mathbf{s}$ by an unshared polynomial is a linear operation so, it can be done on each share separately. The result of the polynomial multiplication is fed to a custom threshold decoder. The decoder uses a masked lookup table. However, to simplify the function calculated by the table, the authors use a set of rules to reduce the number of inputs to the lookup table. The main disadvantage of this decoder is that it increases the decryption failure rate and has a large performance overhead due to the need to repeatedly check the set of rules. The hardware crypto-processor reported in [16] is 20% larger and requires 2.6× more cycles to perform decryption compared to the unprotected design.

Many real-wold applications require the use of schemes that resists chosen-ciphertext attacks (CCA) and adaptive chosen-ciphertext attacks (CCA2). Oder et al. investigated masked implementations for CCA2-secured ring-LWE schemes in [15]. The authors developed a unit (MDecode) that receives the arithmetically

shared polynomial coefficients, converts them to Boolean sharing, and outputs the decoded version. However, their design requires 5.7× more clock cycles compared to the unprotected implementation.

A first-order SCA resistant software implementation of Saber was introduced by Beirendonck et al. in [4], building on work started by Verhulst [19]. The reported overhead of this work is 2.52× in terms of clock cycles compared to the unprotected software. This low overhead is due to Saber's power-of-two moduli and the reliance on rounding for noise generation. A significant contribution of this work is a unit that performs logical shifting on arithmetic shares, based on arithmetic-to-Boolean algorithms by Coron and Debraiz [6, 7]. Their binomial sampler is based on the bit-sliced masked binomial sampler by Schneider et al. [17].

In April 2021, Fritzmann et al. reported a masked SW/HW co-design that supports Saber and Kyber [8]. Their design is based on an open-source RISC-V implementation, in which they added accelerators and instruction-set extensions for PQC algorithms. The accelerators reported are used to speed up hashing, binomial sampling, polynomial multiplication, Arithmetic-to-Boolean (A2B), and Boolean-to-Arithmetic (B2A) operations. The authors report a 2.63× performance overhead for Saber compared to unprotected implementations.

3 Background

3.1 Saber

Saber is a lattice-based KEM that depends on the hardness of the Module Learning With Rounding (MLWR) problem [18]. KEMs use a public and private key pair to generate and securely exchange keys between communication parties. Specifically, Alice generates a key pair, keeps the private key, and distributes the public key. Bob provides Alice's public key to the encapsulation algorithm to generate a secrete key K and ciphertext c. The ciphertext can now be transmitted to Alice. Alice feeds her private key and the ciphertext to the decapsulation algorithm to generate the secret key K.

We concentrate on the SCA protection of the CCA-Secure decapsulation algorithm, Saber.KEM.Decaps, since it uses the long-term private key. The Saber.KEM.Decaps algorithm is based on the CPA-secure algorithms Saber.PKE.Enc and Saber.PKE.Dec. These algorithms are shown in Algorithms 1–3 for reference. Detailed specification can be found at [18].

Saber uses power-of-two moduli, and the primary operation performed is polynomial multiplication. Other significant operations include hashing, an extendable output function, and Binomial sampling.

3.2 Masking

In this work, we utilize masking as an SCA countermeasure. Masking is a well-researched countermeasure that provides a basis for constructing provably secure

Algorithm 1 Saber.PKE.Enc [18]

Require: $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), m \in R_2; r)$
Ensure: $c := (c_m, \boldsymbol{b'})$
1: $A = \mathtt{gen}(seed_{\mathbf{A}}) \in R_q^{l \times l}$
2: $\boldsymbol{s'} = \beta_\mu(R_q^{l \times 1}; r)$
3: $\boldsymbol{b'} = ((\boldsymbol{A}^T \boldsymbol{s'} + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
4: $v' = \boldsymbol{b}^T (\boldsymbol{s'} \bmod p) \in R_p$
5: $c_m = (v' + h_1 - 2^{\epsilon_p - 1} m \bmod p) \gg (\epsilon_q - \epsilon_T) \in R_T$

Algorithm 2 Saber.PKE.Dec [18]

Require: $(\boldsymbol{s}, c := (c_m, \boldsymbol{b'}))$
Ensure: m'
1: $v = \boldsymbol{b}^T (\boldsymbol{s} \bmod p) \in R_p$
2: $m' = ((v - 2^{\epsilon_p - \epsilon_T} c_m + h_2) \bmod p) \gg (\epsilon_p - 1) \in R_2$

Algorithm 3 Saber.KEM.Decaps [18]

Require: $(sk := (z, pkh, pk, \boldsymbol{s}), c)$
Ensure: K
1: $m' = \mathtt{Saber.PKE.Dec}(\mathbf{s}, c)$
2: $(r', \hat{K}') = \mathcal{G}(pkh, m')$
3: $c' = \mathtt{Saber.PKE.Enc}(pk, m'; r')$
4: **if** $c = c'$ **then**
5: $\quad K = \mathcal{H}(\hat{K}, c)$
6: **else**
7: $\quad K = \mathcal{H}(z, c)$
8: **end if**

systems provided that certain assumptions hold. In general, two components define a masking scheme: 1) the method used to split the data into shares, 2) the method used to perform computations on these shares.

For example, in Boolean masking, each sensitive variable x is split into n shares $x_0, x_1, \ldots, x_{n-1}$ such that $\bigoplus x_i = x$. A commonly used way to achieve this is by generating $n-1$ random masks $m_0, m_1, \ldots, m_{n-2}$, setting $x_0 = m_0, x_1 = m_1, \ldots, x_{n-2} = m_{n-2}$, and computing $x_{n-1} = x \oplus m_0 \oplus m_1 \oplus \cdots \oplus m_{n-2}$. On the other hand, in arithmetic masking, a variable a is split into n shares $a_0, a1, \ldots, a_{n-1}$ such that $\sum a_i \mod q = a$. This can be achieved by generating $n-1$ random masks $m_0, m_1, \ldots, m_{n-2}$, setting $a_0 = m_0, a_1 = m_1, \ldots, a_{n-2} = m_{n-2}$, and computing $a_{n-1} = (a - m_0 - m_1 - m_{n-2}) \mod q$.

The computation on the shares should be performed such that all intermediate values are statistically independent of the unshared sensitive variables.

Masking linear functions is trivial. The same function is duplicated, with each instance taking one share of each input variable and producing one share of each output variable. Non-linear functions require much more care to make sure the implementation is correct and secure.

3.3 Domain Oriented Masking

Domain Oriented Masking (DOM) [11], introduced by Gross et al., provides security against SCA attacks in the presence of glitches. It also allows building circuits that can be synthesized for an arbitrary protection order. Similar to classical Boolean masking, variables are split into shares. For example, x is split into x_0 and x_1 such that $x = x_0 \oplus x_1$.

DOM uses the concept of share domains, where every share of each variable is associated with a domain. For example, x_0 and y_0 can be associated with *Domain0*.

In DOM, calculations are done so that data in different domains are kept independent of each other. In case data from two domains must be combined, steps are taken to preserve this independence. Linear functions are trivial to calculate since they require shares from each domain to be used separately. In non-linear functions, however, shares from different domains must be mixed.

4 Methodology

To study the impact of applying SCA countermeasures on the hardware implementations of Saber, we start by developing a baseline lightweight hardware implementation. This allows us to reuse components from the unprotected design, enabling meaningful comparison and evaluation of the cost of protection. At the same time, some components remain unchanged in the protected implementations. We choose a lightweight (LW) implementation because LW applications are especially vulnerable to SCA attacks. In many cases, LW applications have limited or no physical security, allowing easy collection of side-channel information by adversaries. We utilize the Register-Transfer-Level (RTL) methodology to construct our hardware. RTL provides granular control over operations, which simplifies countermeasure application. Additionally, hardware implementations provide performance and power efficiency, which are helpful in many applications. We primarily use VHDL for hardware description, except for the SHA-3 core, which is written using Chisel.

The baseline Saber implementation is then protected against DPA using masking countermeasures, adapting protection schemes to hardware when necessary. Furthermore, we design flexible hardware that has performance and area trade-offs. Doing that results in a highly configurable implementation that can be adapted to a wide range of applications.

The security of our design has been experimentally verified using the Test Vector Leakage Assessment methodology [9]. Finally, we benchmark our design on widely used state-of-the-art FPGA devices to quantify the resource utilization and performance to evaluate the effect of applying the countermeasures on Saber. The results are compared to masked software and software/hardware co-design implementations of Saber.

5 Baseline Lightweight Saber Implementation

The datapath of our hardware implementation of Saber, capable of performing encapsulation and decapsulation, is shown in Figure 1. The figure omits control signals for clarity. The design uses a FIFO-based interface with one input port and one output port. This interface facilitates connecting the design as an accelerator to processors using similar interfaces such as AXI stream [2]. We use memory to store all data, including keys. We choose a memory width of 16 bits to read/write one polynomial coefficient in one clock cycle since the largest coefficient size is 13 bits, and our lightweight units for polynomial arithmetic receive/produce at most one coefficient per clock cycle. All data kept in memory is in byte-string format. This approach allows data to be kept in a compact, memory-saving form. We utilize *width converters* to perform unpacking byte-strings into polynomials before feeding them into arithmetic units and packing the resulting polynomials into byte-strings before memory write-back on the fly. The central control unit implements the sequencing of operations needed to perform encapsulation and decapsulation. The user of the core uses pre-defined opcodes to select one of the two operations.

Data flow from memory to arithmetic units and back to memory, or from memory to SHA3/Sampling units and back to memory. Combining this simple data flow and utilizing width converters simplify our control logic since width converters adjust the width of data with minimal control signals from the central controller, and the simple data flow minimizes control signals to the datapath.

The general operation of the core is as follows: the core pulls input data via the `din` port and interprets the first word as an opcode to select between encapsulation or decapsulation. If encapsulation was selected, the core loads the public key and the random message from the input port and computes the ciphertext and the secret key. If the operation specified in the opcode is decapsulation, the core loads the public key, the private key, and the ciphertext and computes the secret key. In both cases, the `dout` port is used to output results. Below, we discuss the significant units used in the design in detail.

5.1 Polynomial Arithmetic Units

One of the most intensively used operations in Saber and other lattice-based algorithms is polynomial multiplication. Our design goal is to minimize resource utilization of this operation, which comes at the expense of clock cycles.

We developed a flexible schoolbook multiplier and accumulation unit *Poly-MAC* with a configurable rolling factor $ROLL$, which can be set at synthesis time. We define a multiplier with $ROLL = 1$ as a multiplier capable of performing n coefficient multiplications simultaneously. Here, n is the number of coefficients in a polynomial which is equal to 256 in Saber. Our multiplier multiplies $n/ROLL$ coefficients in one clock cycle, and it needs $n \cdot ROLL$ clock cycles to perform the multiplication of two polynomials. Furthermore, it needs roughly $2n$ clock cycles for input and output. This configuration allows us to have a performance-area trade-off yielding a highly flexible design.

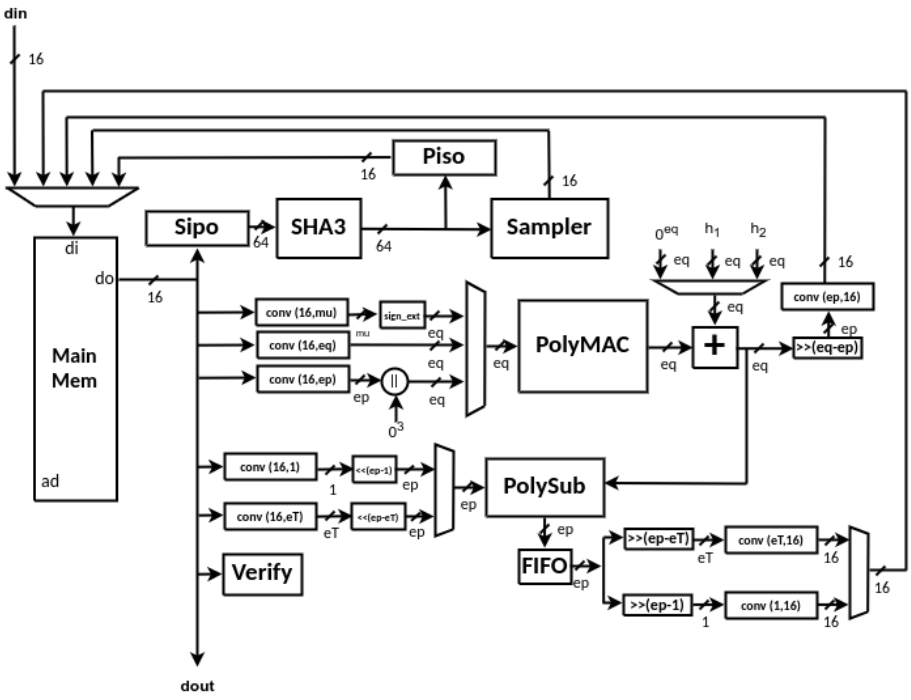


Fig. 1. Lightweight Saber Datapath

The *PolyMAC* unit is shown in Figure 2 and it operates as follows. The multiplier receives the first polynomial *poly1* via the **di** port and stores it internally in a two-dimensional circular input buffer as shown in the left part of Figure 2. The coefficients of *poly1* are organized into columns that can rotate from left to right. *PolyMAC* then receives the second polynomial *poly2* one coefficient at a time via the `di` port and multiplies it by all coefficients of *poly1*. To do the multiplication by all coefficients of *poly1*, the right-most column of the input buffer is multiplied by the current *poly2* coefficient, and the result is stored in the left-most columns of the 2D circular output buffer (shown to the right of the MAC units). The columns of the input and output buffer rotate until all coefficients of *poly1* have been multiplied. The multiplier then pulls the next coefficient of *poly2* until all coefficients are consumed. The result of the polynomial multiplication is stored internally, and the multiplier is ready to output the result or accept another two polynomials to multiply and accumulate to the previous result. This is useful to implement vector-by-vector multiplication. After any multiplication, the result can be cleared using a control signal.

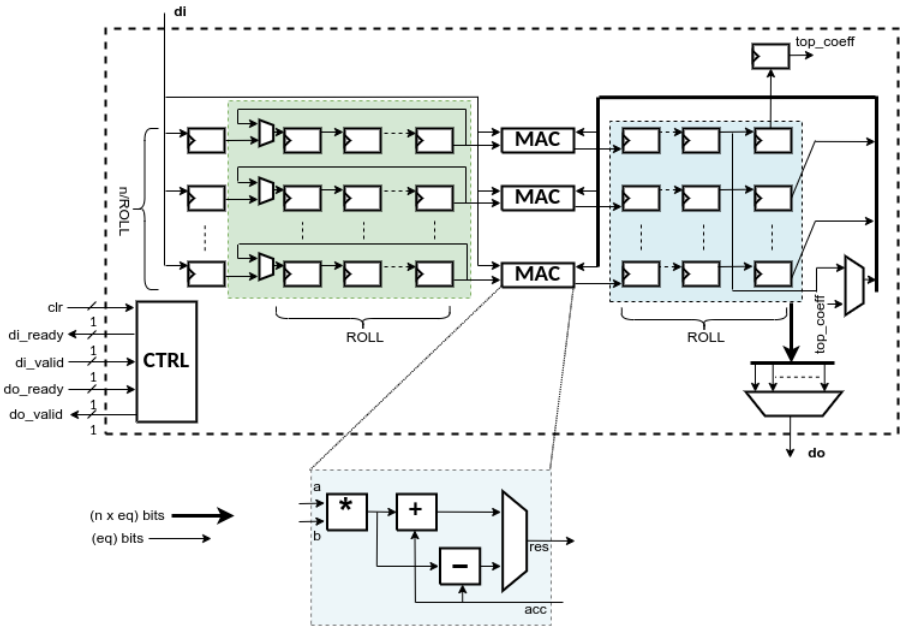


Fig. 2. Configurable Schoolbook Polynomial Multiplier. Input circular buffer highlighted in green and output circular buffer highlighted in blue

The other polynomial arithmetic operation in Saber is polynomial subtraction. This operation is much less time-intensive and has a small effect on the overall execution time of the algorithm. To implement this operation, we developed the *PolySub* unit shown in Figure 3. *PolySub* instantiates a single subtractor capable of subtracting two coefficients at a time. This unit is purely combinational. However, we use control signals for handshaking to make sure that the unit consumes two coefficients from the source before providing the corresponding coefficient of the result at the output. Constants h_1 and h_2 are added using a simple adder at the output of the *PolyMAC* unit, capable of adding two coefficients together in one clock cycle.

5.2 SHA3 Unit

We have developed a flexible SHA-3 unit that can be configured to process a configurable number of state slices to provide performance/area trade-off. Additionally, the IO width of the module is configurable. The core user can select between SHA3-256, SHA3-512, and SHAKE128 functions using a command word. All of these functions are required by Saber. This core has been written in Chisel to exploit its capability to generate highly configurable hardware.

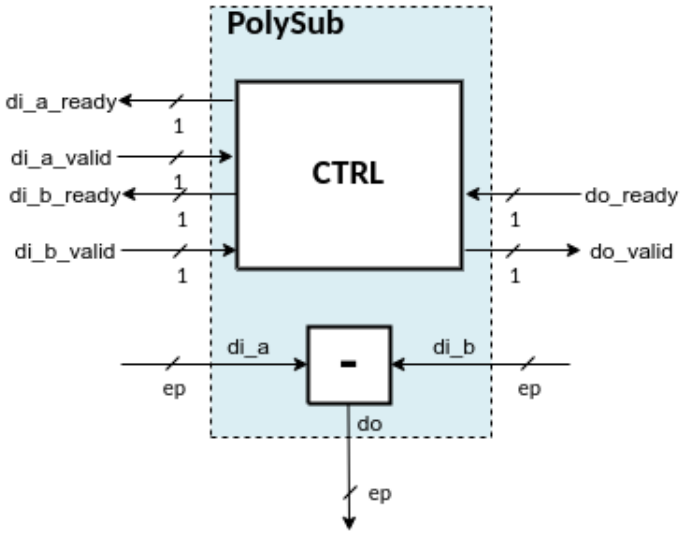


Fig. 3. Polynomial Subtractor

5.3 CBD Sampler

Saber.KEM.Decaps uses Centered Binomial Distribution (CBD) to sample the polynomial vector $\mathbf{s}'$. To generate one binomial sample, our sampler takes two $\mu/2$ bit-wide uniform samples x and y and calculates the CBD sample as $HW(x) - HW(y)$, where $HW(.)$ is the Hamming weight function. Figure 4 shows the sampler unit. It receives 64 bits of uniform randomness generated by SHA-3 and converts it into eight binomial samples in two clock cycles.

5.4 Width Converter Unit

Saber uses many polynomial coefficient sizes. For example, Saber uses polynomials with coefficient sizes of eq, ep, and eT, which are equal to 13, 10, and 4 bits, respectively. To avoid designing separate packing and unpacking units for each size, we developed a flexible width converter with arbitrary input and output width. This unit is essentially an asymmetric FIFO. In Figure 1, width converters are labeled `conv(WI,WO)`, where WI and WO are the input width and output width (in bits), respectively.

Figure 5 shows the internal structure of this unit. We use asymmetric RAM to briefly store the input data and allow it to be read via the output port. Control logic is needed to keep track of pointers to locations for the next read and write and the number of bits stored in the width converter. Utilizing such a unit simplifies data packing and unpacking since the central controller delegates this task to the width converters and only enables the proper width converters

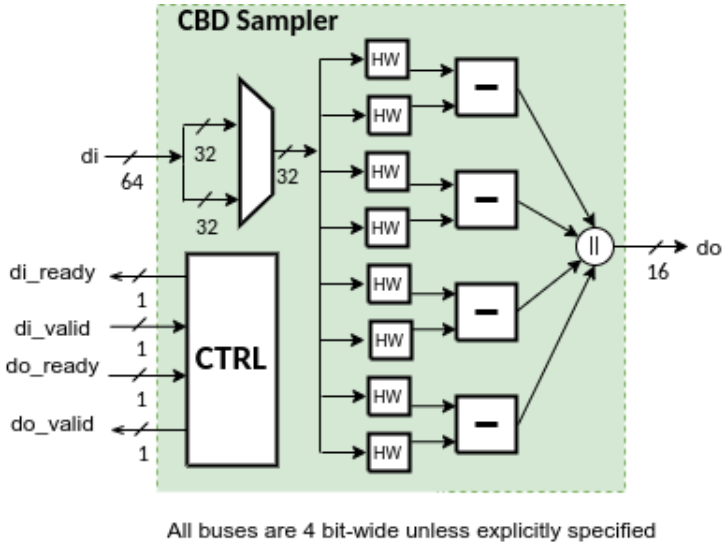


Fig. 4. CBD Sampler

for the current transaction. At the inputs of polynomial arithmetic units, we instantiated width converters to convert from memory width to the coefficient sizes processed by the unit. At the output, we instantiate width converters to pack the data into the memory words on the fly.

5.5 Other Units

The ciphertext verification is done using a comparator that compares two memory locations in two clock cycles. If the contents of the two locations are not equal, we set a flag to indicate the inequality. Regardless of the comparison outcome, we go through all the ciphertext c and the re-encryption ciphertext c' to ensure that our implementation runs in constant time, which is necessary to resist timing attacks. The left-shift operations, which are used for rounding, are free in hardware.

6 Masked Saber Implementation

Contrary to encapsulation, the decapsulation process utilizes the long-term private key, which makes it vulnerable to side-channel analysis. We implement a masked full hardware implementation of Saber.KEM.Decaps based on our lightweight hardware design. We adapt general ideas presented in [4] for hardware. The data flow of our masked Saber.KEM.Decaps is shown in Figure 6. All operations that are dependent on the private key are highlighted in grey. SCA attacks could target any intermediate value processed in these units.

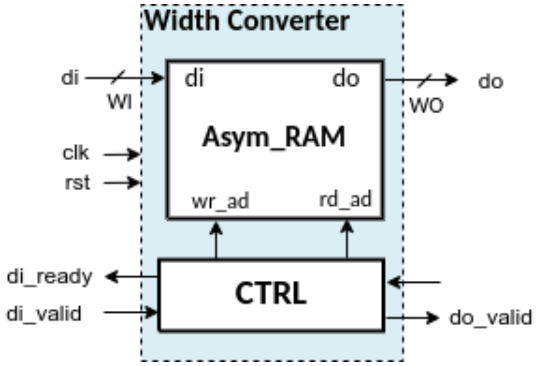


Fig. 5. Width Converter

Polynomial multiplication of an unshared polynomial by a shared polynomial is a linear operation when utilizing arithmetic masking. Hence, multiplication can be done by performing it for each share separately.

Figure 7 depicts the datapath of our masked Saber design. We highlight operations that can be done separately for each of the two shares in green and blue. Hashing using SHA-3, CBD sampling, and rounding include non-linear operations, and both shares mix at some stage in these operations. We highlight these units in red. Eventually, these units produce two shares of data that can be safely consumed in destination domains. In Figure 7, data generally flows from the two memories inward through linear polynomial arithmetic units, then through non-linear rounding units in the center of the figure, and back to main memories. Also, data can flow from the memories to the SHA-3/Sampling units in the middle of the figure and back to memory.

The linear units in the masked design are the same units used in the baseline design. We duplicated these units for each of the two shares. However, non-linear units were re-implemented. We perform constant addition of h_1 and h_2 constants to one of the shares only.

In the following subsections, we describe the hardware implementation of the primary units of the protected design in detail.

6.1 Polynomial Arithmetic Units

Polynomial multiplication is done using the approach used previously by Reparaz et al. in [16]. Since polynomial multiplication is linear for arithmetic masking, secret polynomials are split into two arithmetic shares (coefficient-wise). For a polynomial s, two polynomials s_0 and s_1 are generated such that $s = s_0 + s_1$. Now, multiplication of the shared version of s by another unshared polynomial w is performed as $w*s_0+w*s_1$. Polynomial addition/subtraction of an unshared polynomial is performed on only one share.

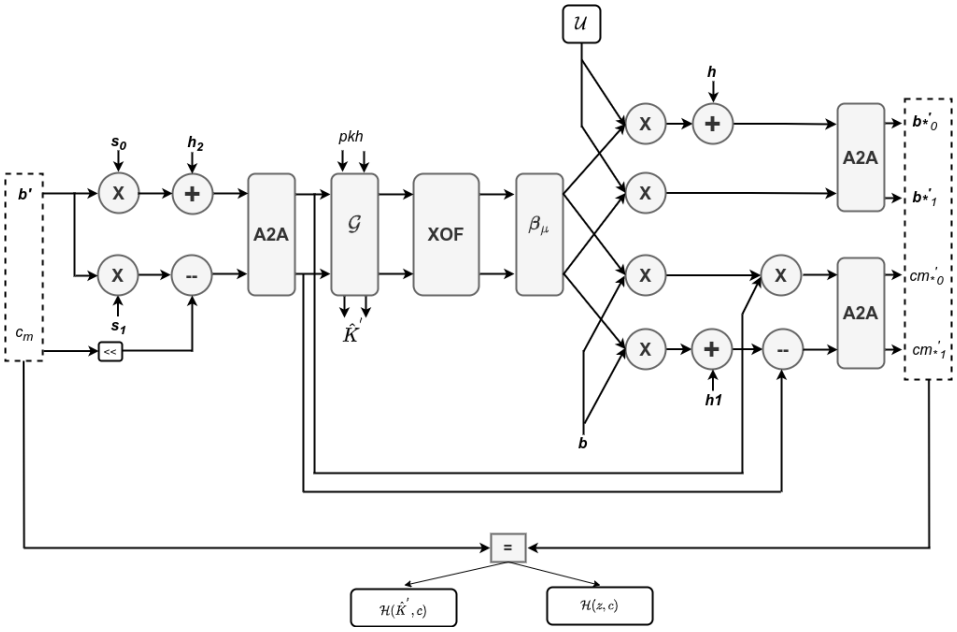


Fig. 6. Masked Saber Decapsulation Data Flow [4]

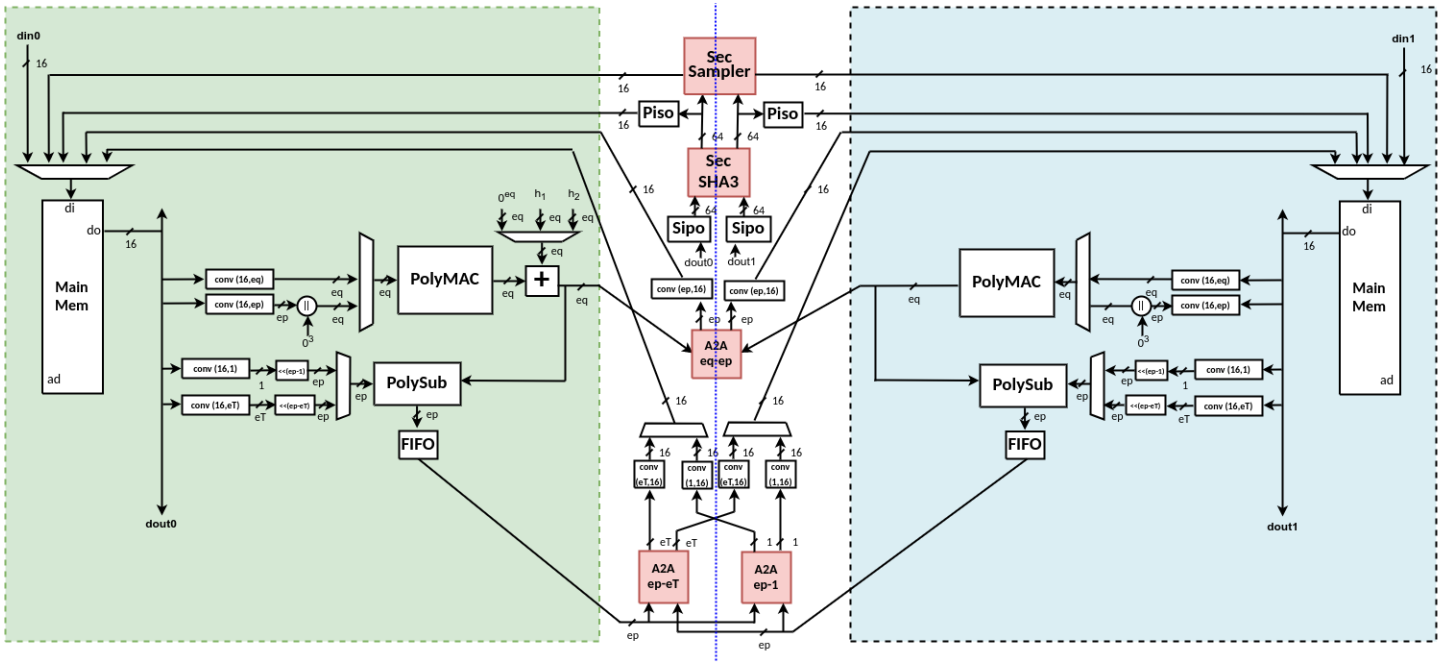


Fig. 7. Protected Saber Datapath. Operations performed on only one share are highlighted in green and blue. Operations that must mix shares (non-linear operations) are highlighted in red.

6.2 SHA3 Unit

We utilize Domain-Oriented Masking (DOM) [11] to develop a first-order protected implementation of our SHA3 core based on [3]. As the input of the Keccak core comes from a uniformly random distribution, we can use uncorrelated state bits to provide for the randomness required for the non-linear χ operation [3].

6.3 CBD Sampler

As shown in Figure 6, the CBD sampler in Saber must be protected against SCA. This sampler should securely compute a CBD sample as the difference between the Hamming weights of two uniform samples x and y as discussed previously.

The masked sampler takes Boolean shares from SHAKE as input. However, the subsequent operations (i.e., polynomial multiplication) use arithmetic shares.

We implemented a masked CBD sampler per Algorithm 4 which was introduced by Schneider et al. [17]. $\mathbf{x}$ and $\mathbf{y}$ are two μ-bit numbers in Boolean sharing representation. The output $\mathbf{A}$ is an arithmetic sharing representation of $HW(x) - HW(y)$, i.e., $\sum A_i = HW(x) - HW(y)$. This task is accomplished by converting one bit at a time of $\mathbf{x}$ and $\mathbf{y}$ from Boolean to arithmetic representation. The generated arithmetic shares are added to compute the Hamming weight of x and y, and finally, a subtraction is performed to compute the binomial sample.

We utilized Goubin's method [10] for Boolean-to-Arithmetic conversion `B2A`. In this conversion, $x = x_p \oplus r$ is converted to arithmetic masking in the form $x = A + r \mod 2^K$ where $K \geq 1$. In this method, the share r is kept as is and A is calculated from the Boolean shares and a random value γ as follows:

$$A = [(x' \oplus \gamma) - \gamma] \oplus x' \oplus [(x' \oplus (r \oplus \gamma)) - (r \oplus \gamma)] \tag{1}$$

Goubin's B2A conversion is efficient and lightweight. Additionally, it works a power-of-two modulus, which makes it suitable for Saber. We used synchronization registers to prevent mixing intermediates that depend on both Boolean shares in our hardware implementation.

The SW/HW design in [8] uses B2A and A2B conversion algorithms from [5]. Both algorithms utilize secure addition over Boolean shares. This enables them to use the same adder to accelerate both operations. In our case, since the sampler is a standalone module, we chose to use Goubin's method since it is suitable for our lightweight design.

Figure 8 depicts our hardware implementation of the CBD sampler. In this figure, we omit control signals and randomness distribution for simplicity. The sampler can work on NS (number of samples) CBD samples at a time. We instantiate NS B2A converters to convert bits from x and a similar number of converters to convert bits from y. The converted shares are then accumulated into registers $A0\_1$ to $A0\_NS$ and $A1\_1$ to $A1\_NS$. Eventually, these samples are sent to the output using parallel-in-serial-out (PISO).

Algorithm 4 SecSampler1 [17]

Require: $\mathbf{x} = (x_i)_{1 \le i \le n} \in \mathbb{F}_{2^\mu}, \mathbf{y} = (y_i)_{1 \le i \le n} \in \mathbb{F}_{2^\mu}$, such that $\bigoplus_i x_i = x$ and $\bigoplus_i y_i = y$

Ensure: $\mathbf{A} = (A_i)_{1 \le i \le n} \in \mathbb{F}_q$ such that $\sum_i A_i = HW(x) - HW(y) \mod q$

1: $(A_i)_{1 \le i \le n} \leftarrow 0$
2: **for** $i = 0$ **to** $\mu - 1$ **do**
3: $\quad (A_i)_{1 \le i \le n} \leftarrow 0$
4: $\quad \mathbf{B} \leftarrow B2A((\mathbf{x} >> i \wedge 1))$
5: $\quad \mathbf{C} \leftarrow B2A((\mathbf{y} >> i \wedge 1))$
6: $\quad \mathbf{A} \leftarrow \mathbf{A} + \mathbf{B} \mod q$
7: $\quad \mathbf{A} \leftarrow \mathbf{A} - \mathbf{C} \mod q$
8: **end for**

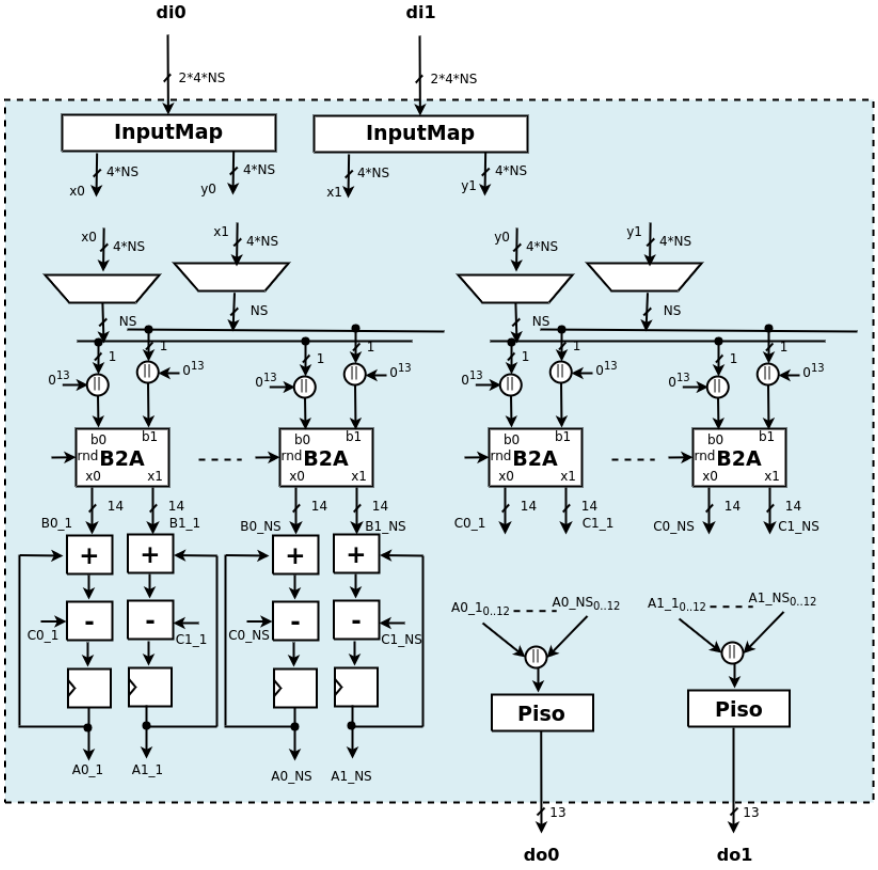


Fig. 8. Masked CBD Sampler

6.4 Masked Logical Shifting

In Saber, noise is introduced into MLWR samples by truncating LSB bits. This operation is free in unprotected hardware. However, in the masked implementation of Saber, this is not as straightforward. This is because the input to this operation consists of arithmetic shares produced by polynomial arithmetic units. However, the logical shift is a Boolean operation. The most straightforward solution to this issue is applying A2B conversion, performing the logic shift on Boolean shares, and using B2A conversion to convert the shares back to arithmetic shares. Many algorithms for B2A and A2B conversion exist. Goubin's B2A conversion [10] is efficient. However, the A2B algorithm proposed in [10] is not as efficient. Coron proposed a table-based method for A2B conversion that can be more efficient than Goubin's method in some cases [6]. A bug in Coron's A2B algorithm was later fixed by Debraiz in [7].

Since the LSB bits are discarded in Saber, it is not efficient to perform all the calculations to convert them into Boolean. The authors of [4] exploited this fact to produce an efficient masked logic shift unit based on [6] and [7]. The authors call this algorithm A2A since it accepts and produces arithmetic shares. This algorithm, adapted from [4] is listed in Algorithm 5.

The A2A logical shift algorithm accepts (A, R) such that $x = A + R \mod s^{m+n\cdot k}$ and returns (A, R) such that $x >> (n \cdot k) = A + R \mod 2^m$, which is the shifted version of x in arithmetic shares. The shifts in Saber are $>> 9, >> 6$ and $>> 3$. Our hardware implementation of the A2A algorithm is shown in Figure 9. We use registers to store the values of the algorithm intermediates. Since the algorithm requires various synchronization stages, we use registers to stop glitch propagation in hardware. We adopt the (m, n, k) values used in [4]. Specifically we set $(m, n, k) = (1, 3, 3), (4, 2, 3)$ and $(10, 1, 3)$ for the $>> 9, >> 6$ and $>> 3$ shifts, respectively. The operation of this module is as follows: first, the module is initialized and it pre-computes the value Γ and the table T. The hardware to compute this step is not shown in Figure 9 for simplicity. Once the module is initialized, it can accept the shares (A, R), and return the shifted version in arithmetic shares via the `Aout` and `Rout` ports.

7 Leakage Assessment

We performed a non-specific fixed-vs-random Test Vector Leakage Assessment (TVLA) [9] to test the first-order leakage of the design. We instantiated the design-under-test (DUT) in the NewAE CW305 target board, which is an Artix-7-based board. The DUT power consumption is measured at the output of the CW305's onboard amplifier, which amplifies the voltage drop across the onboard 0.1 Ω resistor. The DUT was clocked at 12.5 MHz, and a USB3-based oscilloscope (Picoscope 5000) was used to collect traces at a sampling rate of 125 MS/s, and 8-bit sample resolution. We utilized the Flexible Opensource workBench fOr Side-channel analysis (FOBOS) [1] platform to control test-vector communication and trace capture from the oscilloscope. The fixed test vectors are formed by generating fresh sharing of a fixed private key, and the random test vectors are

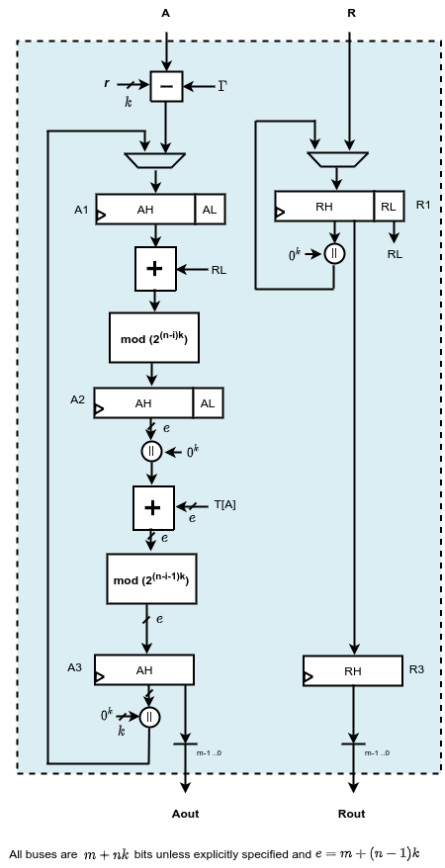


Fig. 9. A2A logical shift unit

generated using a completely random private key. In both cases, the rest of the test vector consists of fixed ciphertext and public key.

To validate our experimental setup, we performed a TVLA test with the PRNG output set to zero. This disables the countermeasures since they depend on randomness generated from the PRNG. The result of this test is shown in Figure 10. As expected, significant leakage is detected. This can be observed even at 2,000 traces.

To test the protected version, we enabled the PRNG to activate the countermeasures. The TVLA result after analyzing 100,000 traces is shown in Figure 11. The right-most spike is related to comparing the hash of the input ciphertext and the ciphertext generated by the re-encryption process. This leakage does not provide any useful side-channel information to an attacker, as discussed in [4].

Algorithm 5 A2A Logical Shift [4]

Require: (A, R) such that $x = A + R \mod 2^{m+n\cdot k}$, T, r, γ
Ensure: (A, R) such that $x >> (n \cdot k) = A + R \mod 2^{m}$
/*Let $A = (A_h||A_l), R = (R_h, R_l)$ where A_l, R_l the k LSB bits.*/
1: $\Gamma \leftarrow \sum_{i=1}^{n} 2^{i\cdot k} \cdot \gamma \mod 2^{m+n\cdot k}$
2: $P \leftarrow \sum_{i=0}^{n-1} 2^{i\cdot k} \cdot r \mod 2^{m+n\cdot k}$
3: $A \leftarrow A - P \mod 2^{m+n\cdot k}$
4: $A \leftarrow A - \Gamma \mod 2^{m+n\cdot k}$
5: **for** $i = 0$ **to** $n - 1$ **do**
6: $A \leftarrow A + R_l \mod 2^{m+(n-i)\cdot k}$
7: $A_h \leftarrow A_h + T[A_l] \mod 2^{m+(n-i-1)\cdot k}$
8: $A \leftarrow A_h$
9: $R \leftarrow R_h$
10: **end for**

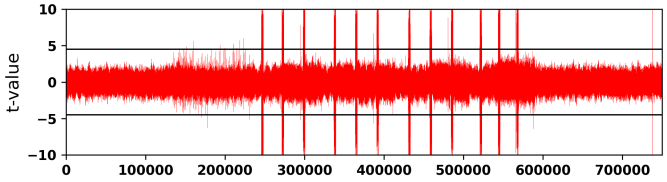


Fig. 10. TVLA result with PRNG disabled (2,000 traces)

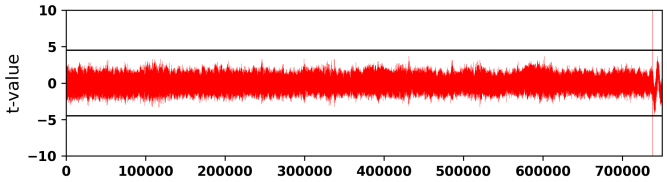


Fig. 11. TVLA Result with PRNG enabled (100,000 traces)

All other points in the TVLA result are below the threshold, indicating that our countermeasures are effective.

Although the protected version shows significant leakage reduction, it can still be vulnerable to fault attacks as well as profiling and deep learning-based attacks such as [14, 13]. We leave protection against these attacks for future work.

8 Results and Comparison

To quantify the cost and performance of our baseline and masked Saber designs, we benchmark them on Xilinx Artix-7 FPGA. Resource utilization in terms of lookup tables (LUTs), flip-flops (FFs), and the number of DSP units is provided. We also provide latency information in clock cycles, maximum frequency, and encapsulation and decapsulation time. These information are shown in Tables 1 and 2.

Saber-r8 refers to our baseline design with $PolyMAC$ rolling factor, ROLL, set to 8, so it can perform $n/8 = 32$ coefficient multiplications in one clock cycle. This is the variant that we report in Tables 1 and 2. Saber-r8 has a low area footprint and requires only 6,713 LUTs and 32 DSPs. On the other hand, Saber-r8-masked, the corresponding masked design, uses 19,299 LUTs and 64 DSPs. That is 2.9 $\times$ more LUTs and exactly 2 $\times$ more DSP units compared to the baseline unprotected variant. Since our baseline design has a small footprint, we decided to duplicate the logic and process shares simultaneously in the masked design. Another option is to use the same hardware resources and process the shares sequentially at the expense of latency. The protected design needs twice as many DSP units because it uses two $PolyMAC$ units, the only unit that uses DSPs.

Our masked design performs decapsulation in 576 μs, assuming keys are already loaded. This is 1.36 $\times$ the baseline unprotected variant.

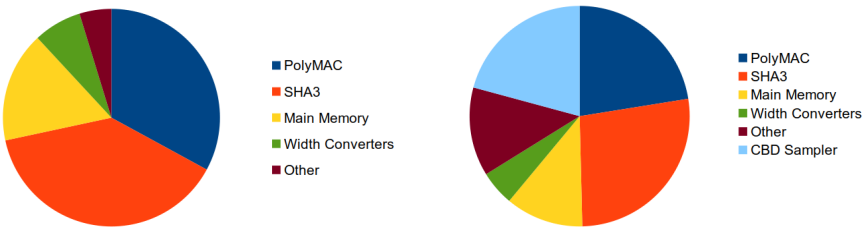
To evaluate how our designs compare to previously reported masked implementations of Saber on various platforms, we summarize all results in Table 1 and 2.

In [4], the authors report a masked software implementation of Saber.KEM.Decaps and benchmarking results on STM32F407-DISCOVERY board featuring an ARM Cortex-M4 processor. The decapsulation time reported is 2,833,348 clock cycles, 2.52$\times$ more than the unprotected decapsulation. For software implementations, it is usual to report cycle count. Execution time can be calculated after knowing the processor clock speed. However, in hardware, the critical path of the design influences the final results, so reporting cycle count and the maximum frequency is helpful. Assuming that the masked software decapsulation in [4] runs at 168MHz, which is the clock frequency used in the STM32F407-DISCOVERY board, protected decapsulation will take 16,865 μs. In this case, our hardware implementation can provide a speedup of 29$\times$.

The SW/HW design reported in [8] is based on an open-source RISC-V implementation augmented with accelerators and instruction-set extensions that can

support Saber and Kyber. The accelerators are used to speed up hashing, binomial sampling, polynomial multiplication, Arithmetic-to-Boolean (A2B), and Boolean-to-Arithmetic (B2A) operations. The authors report 2.63× performance overhead for Saber decapsulation compared to unprotected implementations. In Table 2, we list resource utilization of this SW/HW design. It uses block RAM (BRAMs) while our design does not. However, our designs use more DSP units. In terms of decapsulation time, the protected SW/HW design needs 15,398 μs when run at the reported maximum frequency of 58.8 MHz. Consequently, our full hardware design, Saber-r8-masked, provides a speedup of 26×.

A breakdown of component area (in LUTs) for Saber-r8 and Saber-r8-masked is depicted in Figure 12. The combinations of SHA3, *PolyMAC*, and main memory utilize 88% and 61% for baseline and masked variants, respectively. Width converters that perform packing and unpacking occupy around 7% and 5% in the baseline and masked variants, respectively. In Saber-r8, other components include CBD sampler, *PolySub*, control logic, and other units. These units account for only 4.7%. On the other hand, in Saber-r8-masked, the CBD sampler requires 21% of the LUTs, and other components need 13%. This breakdown shows that further area improvements of both masked and baseline variants will benefit from more area-efficient SHA3 and polynomial multiplication units. A smaller CBD sampler will improve resource utilization of the masked variant.



(a) Baseline Saber-r8 (b) Masked Saber-r8

Fig. 12. Resource Utilization per Unit

9 Conclusions and Future Work

In this work, we report an SCA-resistant hardware implementation of Saber. We have started with a baseline lightweight hardware design and applied side-channel countermeasures to resist DPA attacks. Our masked hardware implementation offers 29× and 26× speedup over previously reported protected software and software/hardware co-design implementations, respectively. Also, our design occupies around 2.9× the number of LUTs and requires 1.4× the latency compared to our baseline design when benchmarked on modern FPGAs. Interesting future work includes investigating resistance against fault and deep

Table 1. Comparison between masked Saber implementations in the literature and designs in this work. Notation: U - unprotected, P - protected.

	Type	**Platform**	**Protection**	**Freq MHz**
This Work	HW	FPGA-Artix7	U	125
				125
			P	125
[4]	SW	ARM Cortex-M4	U	168
			P	168
[8]	SW/HW	RISC-V+ Acc.	U	62.5
				62.5
			P	58.8

Table 2. Comparison between resource utilization and latency of masked Saber implementations in the literature and in this work. Notation: U - unprotected, P - protected.

	Protection	**Resource Utilization**					**Latency**			
		LUTs	**FFs**	**Slices**	**DSPs**	**BRAMs**	**Operation**	**Cycles**	**us**	**ratio**
This Work	U	6,713	7,363	2,631	32	0	Encaps	46,705	373.1	-
							Decaps	52,758	422.1	1.00
	P	19,299	21,977	7,036	64	0	Decaps	72,005	576.0	1.36
[4]	U	-	-	-	-	-	Decaps	1,123,280	6,686.2	1.00
	P						Decaps	2,833,348	16,865.2	2.52
[8]	U	20,697	11,833	6,852	13	36.5	Encaps	308,430	4,934.9	-
							Decaps	347,323	5,557.2	1.00
	P	29,889	17,152	9,641	13	52.5	Decaps	905,395	15,397.9	2.77

learning-based attacks. Reducing resource utilization and improving the performance of hardware implementations of Saber and other finalists in the NIST PQC standardization process will also be helpful.

References

[1] Abubakr Abdulgadir, William Diehl, and Jens-Peter Kaps. "An Open-Source Platform for Evaluation of Hardware Implementations of Lightweight Authenticated Ciphers". In: *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Cancun, Mexico: IEEE, Dec. 2019, pp. 1–5. ISBN: 978-1-72811-957-1. DOI: 10.1109/ReConFig48160.2019.8994788.

[2] ARM. *AMBA AXI Protocol Specification*. https://developer.arm.com/docs/ihi0022/b/amba-axi-protocol-specification-v10. 2003.

[3] Victor Arribas, Begül Bilgin, George Petrides, Svetla Nikova, and Vincent Rijmen. "Rhythmic Keccak: SCA Security and Low Latency in HW". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (Feb. 2018), pp. 269–290. ISSN: 2569-2925. DOI: 10.46586/tches.v2018.i1.269-290.

[4] Michiel Van Beirendonck, Jan-Pieter D'anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. "A Side-Channel-Resistant Implementation of SABER". In: *ACM Journal on Emerging Technologies in*

Computing Systems 17.2 (Apr. 2021), pp. 1–26. ISSN: 1550-4832, 1550-4840. DOI: 10.1145/3429983.

[5] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. "Secure Conversion between Boolean and Arithmetic Masking of Any Order". In: *Advanced Information Systems Engineering*. Ed. by David Hutchison et al. Vol. 7908. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 188–205. ISBN: 978-3-642-38708-1 978-3-642-38709-8. DOI: 10.1007/978-3-662-44709-3_11.

[6] Jean-Sébastien Coron and Alexei Tchulkine. "A New Algorithm for Switching from Arithmetic to Boolean Masking". In: *Cryptographic Hardware and Embedded Systems - CHES 2003*. Ed. by Colin D. Walter, Çetin K. Koç, and Christof Paar. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, pp. 89–97. ISBN: 978-3-540-45238-6. DOI: 10.1007/978-3-540-45238-6_8.

[7] Blandine Debraize. "Efficient and Provably Secure Methods for Switching from Arithmetic to Boolean Masking". In: *Cryptographic Hardware and Embedded Systems – CHES 2012*. Ed. by Emmanuel Prouff and Patrick Schaumont. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 107–121. ISBN: 978-3-642-33027-8. DOI: 10.1007/978-3-642-33027-8_7.

[8] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. *Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography*. Cryptology ePrint Archive 2021/479. Apr. 2021.

[9] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. "A Testing Methodology for Side–Channel Resistance Validation". In: *NIST Non-Invasive Attack Testing Workshop*. Nara, Japan, 2011.

[10] Louis Goubin. "A Sound Method for Switching between Boolean and Arithmetic Masking". In: *Cryptographic Hardware and Embedded Systems — CHES 2001*. Vol. 2162. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 3–15. ISBN: 978-3-540-42521-2 978-3-540-44709-2. DOI: 10.1007/3-540-44709-1_2.

[11] Hannes Gross, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order". In: *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security - TIS'16*. Vienna, Austria: ACM Press, 2016, pp. 3–3. ISBN: 978-1-4503-4575-0. DOI: 10.1145/2996366.2996426.

[12] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Diferential Power Analysis". In: *CRYPTO '99 - 19th International Conference on Cryptology*. Santa Barbara, CA, Aug. 1999.

[13] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. "A Side-Channel Attack on a Masked IND-CCA Secure Saber KEM Implementation". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (Aug. 2021), pp. 676–707. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i4.676-707.

[14] Kalle Ngo, Elena Dubrova, and Thomas Johansson. *Breaking Masked and Shuffled CCA Secure Saber KEM*. Cryptology ePrint Archive 2021/902. July 2021.

[15] Tobias Oder. "Efficient and Side-Channel Resistant Implementation of Lattice-Based Cryptography". Doctoral Thesis. Ruhr-Universität Bochum, Jan. 2020.

[16] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. "Consolidating Masking Schemes". In: *Advances in Cryptology – CRYPTO 2015*. Vol. 9215. LNCS. Santa Barbara, CA: Springer, 2015, pp. 764–783. ISBN: 978-3-662-47988-9 978-3-662-47989-6. DOI: 10.1007/978-3-662-47989-6_37.

[17] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. "Efficiently Masking Binomial Sampling at Arbitrary Orders for Lattice-Based Crypto". In: *Public-Key Cryptography – PKC 2019*. Ed. by Dongdai Lin and Kazue Sako. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 534–564. ISBN: 978-3-030-17259-6. DOI: 10.1007/978-3-030-17259-6_18.

[18] Saber Submission Team. *Round 2 Submissions - Saber Candidate Submission Package*. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/resources.html. Apr. 2019.

[19] Kasper Verhulst. "Power Analysis and Masking of Saber". MA thesis. KU Leuven, 2019.