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.

Abstract. Research in post-quantum cryptography aims to develop and study algorithms that can withstand classical and quantum attacks. The NIST PQC standardization process, now in its third round, specifies ease of protection against side-channel analysis as a desirable selection criterion. In this paper, we report the current status of our work on masked hardware implementation of Saber key encapsulation mechanism, a third-round NIST PQC finalist. We develop a baseline lightweight hardware implementation of Saber and apply side-channel countermeasures. So far, one unit of the design, the sampler, causes information leakage that is being investigated. Since only one unit is to be revised, we provide estimates for the cost and performance of the final design based on the current status. We expect that our protected hardware implementation will be significantly faster than previously reported protected software and software/hardware co-design implementations, respectively. Additionally, we expect that applying side-channel countermeasures to our baseline design will incur approximately 3× 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, including RSA and ECC. This motivates Post-quantum Cryptography (PQC) research and development, aiming to produce algorithms that can withstand quantum and classical attacks. The NIST PQC standardization process, currently in its third round, aims to coordinate the development and analysis of PQC algorithms to eventually select algorithms to be the PQC standard.

Side-channel analysis (SCA), including Differential Power Analysis (DPA) [19], 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 in candidates. Among the most urgent tasks are developing SCA-resistant implementations of third-round finalists and assessing the 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 software, while glitches 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 [5, 6] and software/hardware co-design [10]. In this work, we build on previous work to develop and evaluate SCA-resistant full hardware implementations of Saber. Our design is in its final development stages, and its security against SCA is verified except for one unit, the binomial sampler, which is being revised. We expect that our final hardware design to be significantly faster compared to SW and SW/HW implementations. Additionally, we expect the masked design to use approximately 3× the lookup tables (LUTs) while incurring 1.4× performance penalty compared to the unprotected baseline design when implemented in Xilinx Artix7 FPGAs.

2 Previous Work

PQC algorithm side-channel resistance is an active research field with several open problems. Among the most pressing short-term tasks is developing efficient countermeasures suitable for PQC algorithms and assessing the comparative cost of protecting NIST PQC third-round candidates. The community has made progress towards these goals. In [26], Reparaz et al. proposed a masked implementation for ring-Learning-With-Error (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 simplify the input 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 repeated checking of the rules. The hardware crypto-processor reported in [26] is 20 % larger than the unprotected baseline implementation and requires 2.6× the cycles to perform decryption compared to the unprotected design.

In [25], Reparaz et al. presented a method to develop SCA-protected ring-LWE implementations without using the custom decoder [25], utilizing the additively-homomorphic property of ring-RWLE. In this scheme, a ciphertext mask is calculated by encrypting a random message, then the mask is added to the ciphertext. This randomizes the ciphertext, deprives the attacker of the ability to control ciphertext, and hampers the ability to guess the intermediate values. This inability to guess intermediates impedes direct DPA attacks. However, the authors do not claim theoretical first-order security since the key is not masked.

Oder et al. investigated masked implementations for CCA2-secured ring-LWE schemes in [24]. Many real-wold applications require the use of schemes that resists chosen-ciphertext attacks (CCA) or adaptive chosen-ciphertext attacks (CCA2). The authors developed a unit (MDecode) that receives the arithmetically shared polynomial coefficients, convert them to Boolean and output the decoded version. However, their design comes at a price of 5.7× the clock cycles compared to the unprotected design.

Several side-channel-resistant implementations of PQC algorithms have been reported. In [18], a side-channel resistant crypto-processor that supports NewHope-NIST, NewHope-USENIX, and HILA5 was introduced. Masked GLP, qTESLA, and Dilithium software were developed in [4, 22, 12].

In June 2020, a first-order SCA resistant software implementation of Saber was introduced by Beirendonck et al. in [5], building on work started by Verhulst [31]. 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 the use of 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

Table 1: Saber Notation

Notation	Meaning
$\mathbb{Z}_q$	Ring of integers mod q
$\mathbb{R}_q$	Quotient ring $\mathbb{Z}_q[X]/[X^n + 1]$ where $n = 256$ for Saber and all coefficients $\in \mathbb{Z}_q$
$R^{l \times k}$	An $l \times k$ matrix with each element $\in R$
bold lowercase letter (e.g $\mathbf{v}$)	polynomial vector
bold uppercase letter (e.g $\mathbf{A}$)	polynomial matrix
$x \leftarrow \chi$	Sampling x from distribution χ
$\mathbf{X} \leftarrow \chi(R_q^{l \times k})$	Sampling matrix $\mathbf{X} \in R_q^{l \times k}$, where all coefficients of elements of $\mathbf{X}$ are sampled form χ
$\mathbf{X} \leftarrow \chi(R_q^{l \times k}, r)$	Same as above but here sampling is done deterministically using seed r
$\mathcal{U}$	Uniform distribution
β_μ	Centered binomial distribution with parameter μ. Samples are in the interval $[\frac{\mu}{2}, \frac{-\mu}{2}]$
$\lll$	Integer shift left. Applied coefficient-wise when used with polynomials and matrices
$\ggg$	Integer right left. Applied coefficient-wise when used with polynomials and matrices

by Coron and Debraiz [8, 9]. Their binomial sampler is based on the bitsliced masked binomial sampler by Schneider et al. [28].

In April 2021, Fritzmann et al. reported a masked SW/HW co-design that supports Saber and Kyber. 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 [29] is a Key Encapsulation Mechanism (KEM), currently a NIST PQC third round finalist. Saber is a lattice-based scheme that depends on the hardness of the Module Learning With Rounding (MLWR) problem.

KEMs use a public and private key pair to generate and securely exchange session keys. Specifically, Alice first generates the key pair, keeps the private key and distributes the public key. Bob can use Alice's public key to generate a secrete key K and ciphertext c. The ciphertext can now be transmitted to Alice. Alice uses her private key to decrypt the ciphertext and generate the secret key K. In Saber, there is a small probability of decryption failure.

Table 1 shows the notation used to describe Saber component algorithms, and Table 2 summarizes the parameters of Saber. Please note that we only list the parameters of the Saber variant, i.e., the variant with security level 3. The parameters of the other two variants, namely, LightSaber and FireSaber, are omitted.

Table 2: Saber Parameters. Values listed for the Saber variant only.

Parameter	Purpose	Value
n	The degree of polynomial ring $\mathbb{Z}_q[X]/[X^n+1]$	256
l	The module rank	3
q, p, T	Scheme moduli	$2^{13}, 2^{10}, 2^4$
μ	Centered binomial distribution parameter used for the secret polynomials $\mathbf{s}$ and $\mathbf{s}'$	8
$\mathcal{F}$, $\mathcal{G}$	Hash function	SHA3-256
$\mathcal{H}$	Hash function	SHA3-512
gen	Extendable output function used to generate matrix $\mathbf{A}$	SHAKE128

3.1.1 Saber Public Key Encryption Scheme

Three algorithms, Saber.PKE.KeyGen, Saber.PKE.Enc and Saber.PKE.Dec, constitute the Saber.PKE scheme. Saber.PKE.KeyGen generates a public key pk and a private key sk. Saber.PKE.Enc receives the public key pk and a 256-bit message m and produces ciphertext c. The decryption algorithm Saber.PKE.Dec takes the private key sk and ciphertext c and produces a message m', which, with high probability, is equal to the original message m. Saber.PKE.KeyGen, Saber.PKE.Enc and Saber.PKE.Dec are shown in detail in Algorithms 1, 2 and 3.

Algorithm 1 Saber.PKE.KeyGen() [29]

Require: None
Ensure: $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), \boldsymbol{s})$
1: $seed_{\mathbf{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
2: $A = \mathtt{gen}(seed_{\mathbf{A}}) \in R_q^{l \times l}$
3: $r = \mathcal{U}(\{0,1\}^{256})$
4: $\boldsymbol{s} = \beta_\mu(R_q^{l \times 1}; r)$
5: $\boldsymbol{b} = ((\boldsymbol{A}^T \boldsymbol{s} + \boldsymbol{h}) \bmod q \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$

Algorithm 2 Saber.PKE.Enc [29]

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: **if** *r not specified* **then**
3: $r = \mathcal{U}(\{0,1\}^{256})$
4: **end if**
5: $\boldsymbol{s}' = \beta_\mu(R_q^{l \times 1}; r)$
6: $\boldsymbol{b}' = ((\boldsymbol{A}^T \boldsymbol{s}' + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
7: $v' = \boldsymbol{b}^T (\boldsymbol{s}' \bmod p) \in R_p$
8: $c_m = (v' + h_1 - 2^{\epsilon_p - 1} m \bmod p) \gg (\epsilon_q - \epsilon_T) \in R_T$

3.1.2 Saber Key Encapsulation Mechanism

Built on top of Saber.PKE, Saber Key Encapsulation Mechanism (KEM) generates a session key for two communicating parties. The first party uses the public key to generate a secret key and ciphertext transmitted to the receiver. The receiver uses her private key to decapsulate the secret key from the ciphertext.

Algorithm 3 Saber.PKE.Dec [29]

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

Three algorithms are used in Saber.KEM. Saber.KEM.KeyGen generates a public key pk and a private key sk. Saber.KEM.Encaps takes the public key and produces a secret key K and ciphertext c. Saber.KEM.Decaps takes the ciphertext and the private key and generates the secret key K with high probability. Saber.KEM.KeyGen, Saber.KEM.Encaps and Saber.KEM.Decaps are shown in detail in Algorithms 4, 5 and 6.

Algorithm 4 Saber.KEM.KeyGen [29]

Require: None
Ensure: $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), sk := (z, pkh, pk, \boldsymbol{s}))$
1: $(seed_{\mathbf{A}}, \boldsymbol{b}, \boldsymbol{s}) =$ `Saber.PKE.KeyGen()`
2: $pk = (seed_{\mathbf{A}}, \boldsymbol{b})$
3: $pkh = \mathcal{F}(pk)$
4: $z = \mathcal{U}(\ \{\ 0{,}1\ \}^{256})$

Algorithm 5 Saber.KEM.Encaps [29]

Require: $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}))$
Ensure: (c, K)
1: $m = \mathcal{U}(\ \{\ 0{,}1\ \}^{256})$
2: $(\hat{K}, r) = \mathcal{G}(\mathcal{F}(pk), m)$
3: $c =$ `Saber.PKE.Enc`$(pk,\ m;\ r)$
4: $K = \mathcal{H}(\hat{K}, c)$

3.2 Side-channel Analysis Countermeasures

3.2.1 Masking

Generally, there are two approaches to mitigate leakage from cryptographic devices. The first tries to break the statistical dependence between the intermediate values processed in the implementation and the unshared secret values. This method is called *masking*. Masking is a well-researched countermeasure, which is a solid basis to construct provably secure systems assuming that certain assumptions hold. The other method, called hiding, strives to make the device's power consumption independent of the intermediate values that it processes. This independence is usually achieved by trying to design devices with constant or random power consumption. Hiding has been attempted using the Dynamic Dual Rail [30] techniques, in which two copies of a circuit calculate complementary data so that the sum of their power consumption is constant. However, this method requires the complementary paths to be balanced, which is a challenging task.

In this work, we concentrate on masking countermeasures. 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 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 $x_0 = m_0, x_1 = m_1, \ldots, x_{n-2} = m_{n-2}$, followed

Algorithm 6 Saber.KEM.Decaps [29]

Require: $(sk := (z, pkh, pk, \boldsymbol{s}))$
Ensure: K
1: $m' = \texttt{Saber.PKE.Dec}(\mathbf{s}, c)$
2: $(\hat{K}', r') = \mathcal{G}(pkh, m')$
3: $c' = \texttt{Saber.PKE.Enc}(pk, m'; r')$
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**

by 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 $a_0, a_1, \ldots, a_{n-2}$ and setting $a_0 = m_0, a_1 = m_1, \ldots, a_{n-2} = m_{n-2}$, followed by computing $a_{n-1} = (a - m_0 - m_1 - m_{n-2}) \mod q$.

The function to be masked is also split into component functions. The computation should be performed such that all intermediate values are statistically independent of the unshared sensitive variables.

For linear functions, performing the calculation on shares 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 work to make sure the implementation is correct and secure.

Masked implementations are typically analyzed in a theoretical model introduced by Ishai, Sahai, and Wagner [17] called the probing model. A circuit is secure in the d-probing model if an adversary with d probing needles that can be used to read any d signals in the circuit cannot gain any information about the intermediate value. For example, if we split a variable into $d+1$ Boolean shares, an attacker having access to up to d shares does not learn anything about the original unshared value.

It was shown in the literature that security against d-probing attack is related to security against the dth-order SCA attack. However, in hardware implementations, signal delays vary, causing glitches. Glitches are temporary changes in a digital signal before it stabilizes at the end of a clock cycle. This phenomenon occurs due to asymmetric delays in combinational paths. It was shown in [20, 21] that glitches can depend on unshared data causing leakages in CMOS technology, which is the primary technology used to realize hardware circuits.

3.2.2 Domain Oriented Masking

Domain Oriented Masking (DOM) [15] 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 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. This results in a highly configurable implementation that can be adapted to a wide range of applications.

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 facilitates connecting the design as an accelerator to processors using interfaces such as AXI stream [2] and similar interfaces. We use memory to store all data, including the public and private 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 in one clock cycle. All data kept in memory is in byte-string format. This 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 takes care of implementing the sequence 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. The combination of 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 major units used in the design in detail.

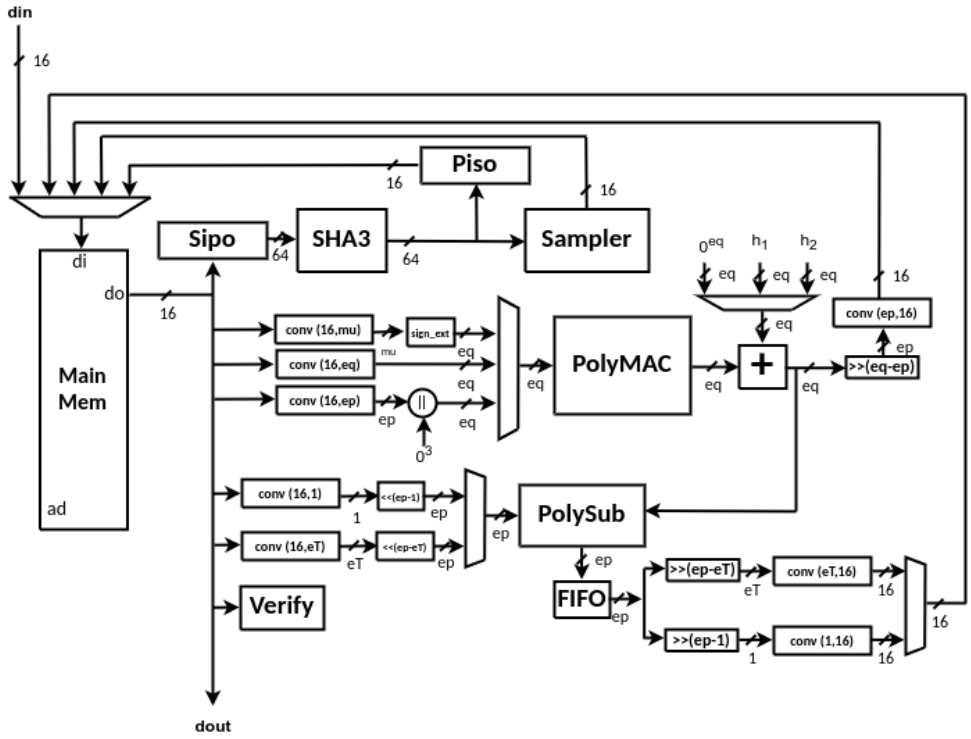


Figure 1: Lightweight Saber Datapath

5.1 Polynomial Arithmetic Units

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

We developed a flexible schoolbook multiplier and accumulation unit *PolyMAC* 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. 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.

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

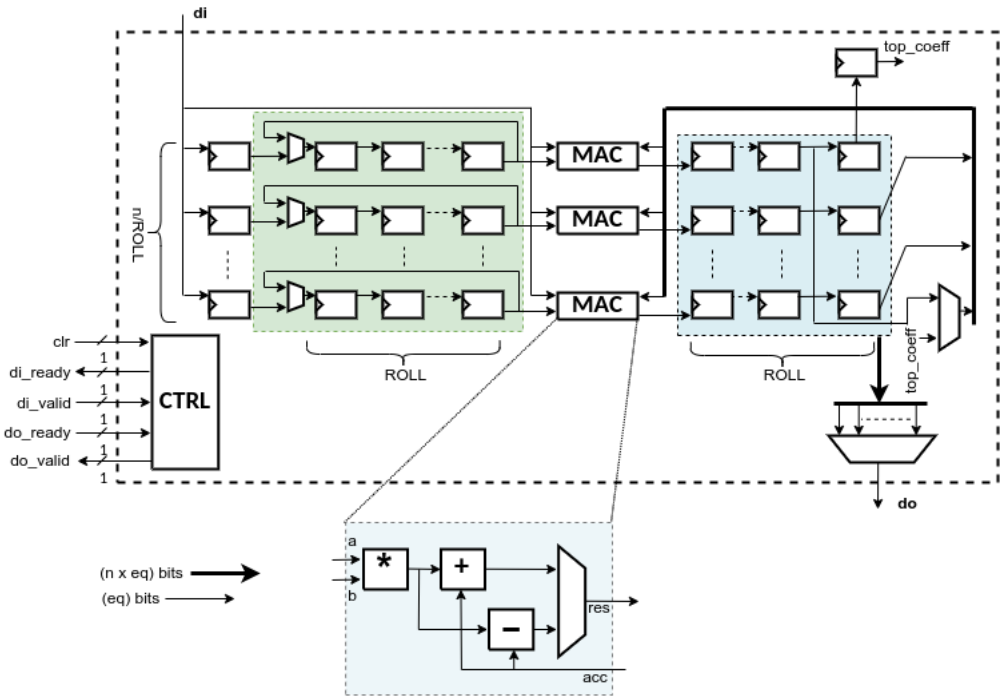


Figure 2: Configurable Schoolbook Polynomial Multiplier. Input circular buffer highlighted in green and output circular buffer highlighted in blue

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.

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.

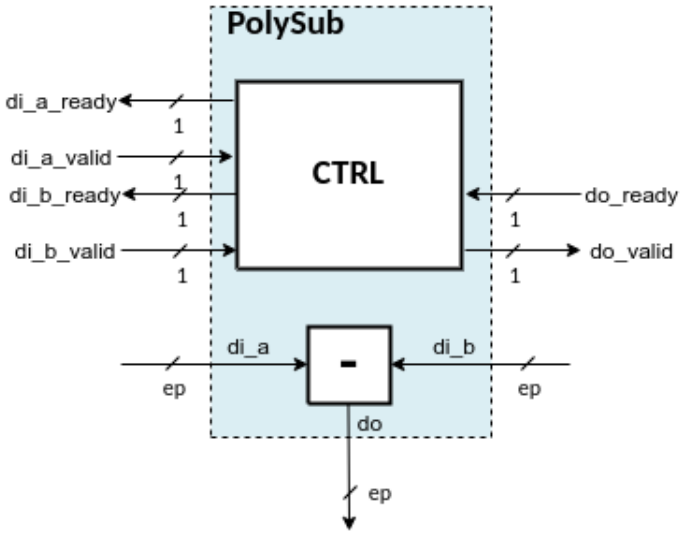


Figure 3: Polynomial Subtractor

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, polynomials with coefficient sizes of eq, ep, and eT are used. 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 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

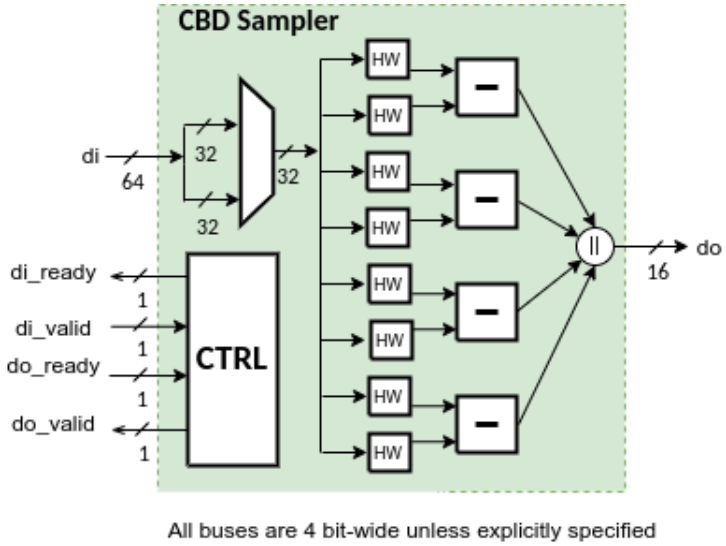


Figure 4: CBD Sampler

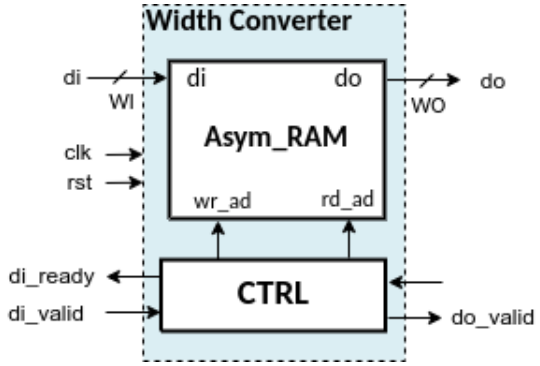


Figure 5: Width Converter

ciphertext c and the re-encryption ciphertext c' to make sure 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 adopt ideas from the masked software implementation reported in [5] and the hardware-software co-design reported in [11] and adapt these schemes for our lightweight hardware design. The data flow of the masked Saber.KEM.Decaps adapted from [5] 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.

Polynomial multiplication of an unshared polynomial by a shared polynomial is a linear operation when utilizing arithmetic masking. The 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.

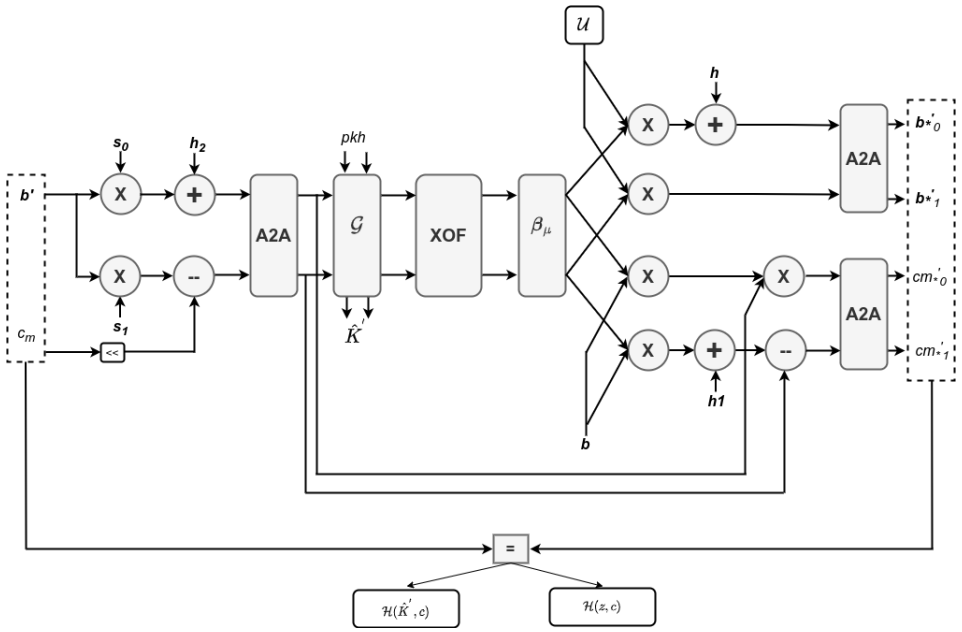


Figure 6: Masked Saber Decapsulation Data Flow [5]

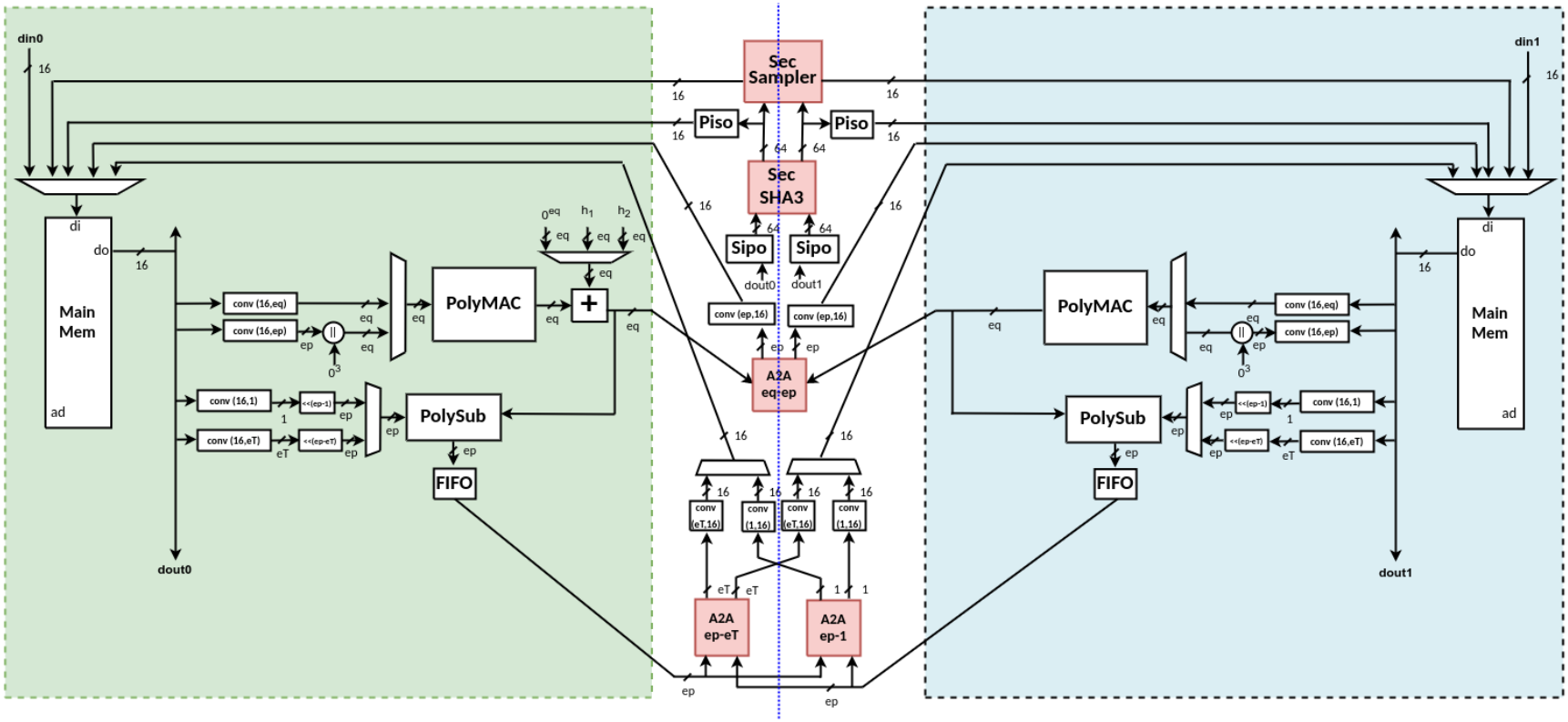


Figure 7: Protected Saber Datapath

6.1 Polynomial Arithmetic Units

Polynomial multiplication is done using the approach used previously by Reparaz et al. in [26]. 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 is 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.

6.2 SHA3 Unit

We utilize Domain-Oriented Masking (DOM) [16] to develop a first-order protected implementation of our SHA3 core based on [3]. As the input 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 based on ideas from [11], which introduces accelerators for the bitsliced sampler described in [28].

Figure 8 depicts our sampler design which computes eight samples in parallel. An adder tree is used to compute $HW(x)+z$, where x and z are in the form of Boolean shares. The tree comprises an array of half adders that use DOM AND gates to compute the carry. The first adder tree is used to compute $HW(y)$ since its z input is set to a shared representation of zero. The negation of $HW(y)$ is subsequently computed by evaluating its 2's complement. Another adder tree takes the negation of $HW(y)$ and x and computes $HW(x) - HW(y)$. Since the result produced by the adder tree is in Boolean shares, a final B2A conversion is performed. The B2A conversion uses the same algorithm as [11] which is originally introduced in [7]. To implement the secure addition required for the B2A conversion, we chose to utilize the protected ripple carry adder (RCA) described in [27] which uses Threshold Implementation (TI) [23]. RCA uses fewer resources at the expense of delay. We use this adder in the B2A algorithm and also to negate $HW(y)$.

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 direct 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 [14] is efficient. However, the A2B algorithm proposed in [14] 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 [8]. A bug in Coron's A2B algorithm was later fixed by Debraiz in [9].

Since the LSB bits are discarded in Saber, it is not efficient to perform all the calculations to convert them into Boolean. [5] exploited this fact to produce an efficient masked logic

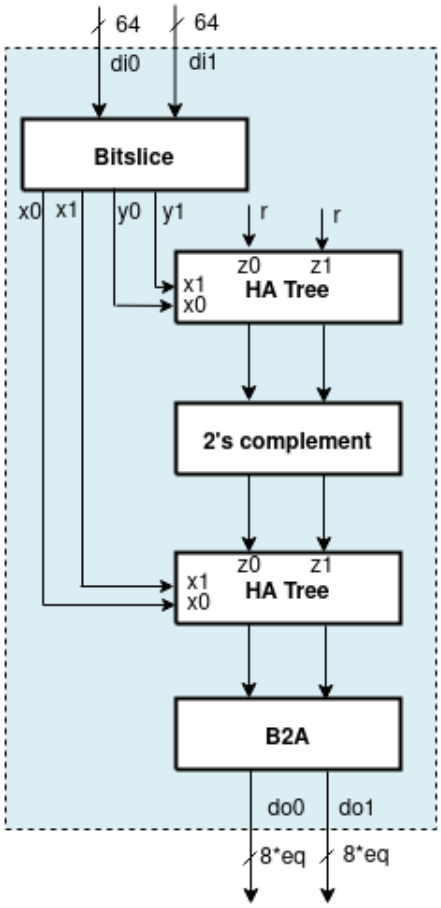


Figure 8: Masked CBD Sampler

shift unit based on [8] and [9]. The authors call this algorithm A2A since it accepts and produces arithmetic shares. This algorithm, adapted from [5] is listed in Algorithm 7.

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 require various synchronization stages, we keep the register that stops glitch propagation in hardware. We adopt the (m, n, k) values used in [5]. 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 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 fixed-vs-random Test Vector Leakage Assessment (TVLA) [13] to test the first-order leakage of the design. We instantiated the design-under-test (DUT) in

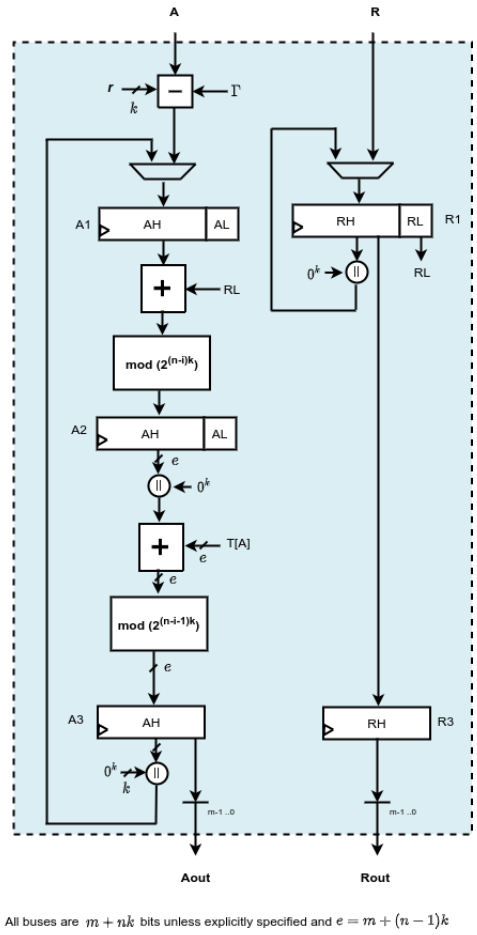


Figure 9: A2A logical shift unit

the NewAE CW305 target board, which is an Artix7-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 1.25 MHz, and a USB3-based oscilloscope (Picoscope 5000) was used to collect traces at a sampling rate of 15.6 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 traces are generated by generating fresh sharing of a fixed private key, and the random traces are generated using a completely random private key. The rest of the test vector consists of fixed ciphertext and public key.

The TVLA result after analyzing 100 thousand traces is shown in Figure 10. 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 [5]. However, few leakage points that correspond to the operation of the CBD sampler exist in the figure. These points are under investigation to identify their root cause.

Algorithm 7 A2A Logical Shift [5]

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**

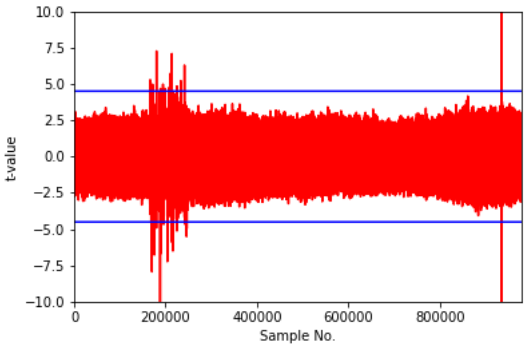


Figure 10: TVLA Result (100,000 traces)

8 Results and Comparison

As mentioned previously, our CBD sampler implementation causes TVLA results to exceed the threshold, indicating leakage. We are investigating this issue, and we expect the design of the sampler to change. However, since only one unit will be changed, we can provide estimates for area and latency based on our current design. We expect these estimates to be close to the fully protected design. To quantify the cost and performance of our baseline and masked Saber designs, we benchmark them on Xilinx Artix7 FPGA. Resource utilization in terms of lookup tables (LUTs), flip-flops (FFs), and the number of DSP units is provided in Table 8. We also provide latency information in clock cycles, maximum frequency, and encapsulation and decapsulation time. 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.

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 19,783 LUTs and 64 DSPs. That is 2.95 × the LUTs and exactly 2 × the 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 the DSP units because it uses two $PolyMAC$ units, the only unit that uses DSPs.

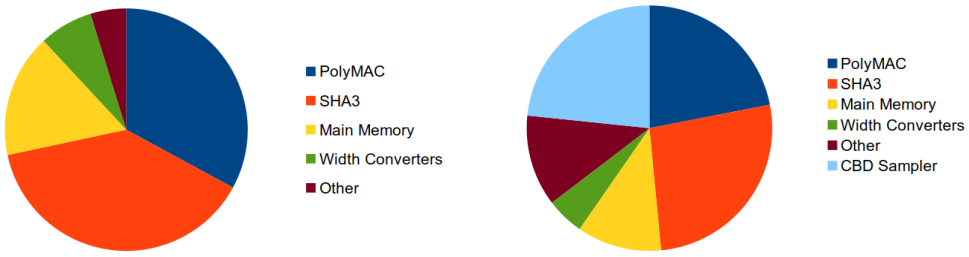
Our masked design performs decapsulation in 739 μs assuming keys are already loaded. This is 1.4 $\times$ the baseline unprotected variant.

To evaluate how our designs compare to previously reported implementations of Saber on various platforms, we listed results from [5] and [11], where SW and SW/HW implementations were reported, and [32], where a hardware implementation is proposed.

[5] reports 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$\times$ their 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 end results, so cycle count and maximum frequency are useful. Assuming that the masked software decapsulation in [5] 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 23$\times$.

The SW/HW design reported in [10] 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$\times$ performance overhead for Saber decapsulation compared to unprotected implementations. In Table 8, 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 21$\times$.

A breakdown of component area (in LUTs) for Saber-r8 and Saber-r8-masked is depicted in Figure 11. The combinations of SHA3, *PolyMAC*, and main memory utilize 88% and 60% 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 23% of the LUTs, and other components need 12%. 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

Figure 11: Resource Utilization per Unit

Table 3: Comparison between Saber implementations in the literature and estimated results of designs in this work (TW)

Algorithm	Type	Platform	Protection	Freq	Resource Utilization						Latency		
				MHz	LUTs	FFs	Slices	DSPs	BRAMs	Operation	Cycles	us	ratio
Saber-r8 [TW]	HW	FPGA-Artix7	unprotected	100	6,713	7,363	2,631	32	0	Encaps	46,705	467.1	-
				100						Decaps	52,758	527.6	1.00
Saber-r8-masked [TW]	HW		Protected	100	19,783	21,576	7,143	64	0	Decaps	73,851	738.5	1.40
Saber [32]	HW	FPGA-UltraScale+	unprotected	100	34,886	9,858	-	85	6.0	Encaps	1,396	14.0	-
			unprotected	100						Decaps	1,684	16.8	-
Saber [5]	SW	ARM Cortex-M4	unprotected	168	-	-	-	-	-	Decaps	1,123,280	6,686.2	1.00
	SW		Protected	168						Decaps	2,833,348	16,865.2	2.52
Saber [11]	SW/HW	RISC-V+ Acc.	unprotected	62.5	20,697	11,833	6,852	13	36.5	Encaps	308,430	4,934.9	-
				62.5						Decaps	347,323	5,557.2	1.00
	SW/HW		Protected	58.8	29,889	17,152	9,641	13	52.5	Decaps	905,395	15,397.9	2.77

9 Conclusions and Future Work

In this work, we report the status of our work on SCA-resistant hardware implementation of Saber. We have started with a baseline lightweight hardware design and applied side-channel countermeasures to resist DPA attacks. So far, the TVLA result exceeds the threshold when the CBD sampler operates, indicating a leakage. This leakage is being investigated, and we expect to revise the sampler implementation. Since only one unit is to be changed, we provide estimates of the resource utilization of our final design. Our masked hardware implementation is expected to offer 23× and 21× speedup over previously reported protected software and software/hardware co-design implementations, respectively. Also, we expect that our final design will occupy around 3× the number of LUTs and require 1.4× the latency compared to our baseline design when benchmarked on modern FPGAs. Future work will include investigating methods to enhance security, reduce resource utilization, and improve the performance of hardware implementations of Saber and other finalists in the NIST PQC standardization process.

References

[1] Abubakr Abdulgadir, William Diehl, and Jens-Peter Kaps. *An Open-Source Platform for Evaluating Side-Channel Countermeasures in Hardware Implementations of Lightweight Authenticated Ciphers*. en. Tech. rep. Nov. 2019, p. 12.

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

[3] Victor Arribas et al. *Rhythmic Keccak: SCA Security and Low Latency in HW*. Tech. rep. 1193. 2017.

[4] Gilles Barthe et al. "Masking the GLP Lattice-Based Signature Scheme at Any Order". en. In: *Advances in Cryptology – EUROCRYPT 2018*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10821. Cham: Springer International Publishing, 2018, pp. 354–384. ISBN: 978-3-319-78374-1 978-3-319-78375-8. DOI: 10.1007/978-3-319-78375-8_12.

[5] Michiel Van Beirendonck et al. *A Side-Channel Resistant Implementation of SABER*. Cryptology ePrint Archive 2020/733. June 2020.

[6] Michiel Van Beirendonck et al. "A Side-Channel-Resistant Implementation of SABER". en. 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.

[7] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. "Secure Conversion between Boolean and Arithmetic Masking of Any Order". en. 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.

[8] Jean-Sébastien Coron and Alexei Tchulkine. "A New Algorithm for Switching from Arithmetic to Boolean Masking". en. 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.

[9] Blandine Debraize. "Efficient and Provably Secure Methods for Switching from Arithmetic to Boolean Masking". en. 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.

[10] Tim Fritzmann, Georg Sigl, and Johanna Sepulveda. "Extending the RISC-V Instruction Set for Hardware Acceleration of the Post-Quantum Scheme LAC". en. In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Grenoble, France: IEEE, Mar. 2020, pp. 1420–1425. ISBN: 978-3-9819263-4-7. DOI: 10.23919/DATE48585.2020.9116567.

[11] Tim Fritzmann et al. *Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography*. Cryptology ePrint Archive 479. Apr. 2021.

[12] François Gérard and Mélissa Rossi. "An Efficient and Provable Masked Implementation of qTESLA". en. In: *Smart Card Research and Advanced Applications* 11833 (2020). Ed. by Sonia Belaïd and Tim Güneysu, pp. 74–91. DOI: 10.1007/978-3-030-42068-0_5.

[13] Gilbert Goodwill et al. "A Testing Methodology for Side–Channel Resistance Validation". In: *NIST Non-Invasive Attack Testing Workshop*. Nara, Japan, 2011.

[14] Louis Goubin. "A Sound Method for Switching between Boolean and Arithmetic Masking". en. In: *Cryptographic Hardware and Embedded Systems — CHES 2001*. Vol. 2162. 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.

[15] Hannes Gross, Stefan Mangard, and Thomas Korak. *Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order*. Cryptology ePrint Archive 2016/486. Nov. 2016.

[16] Hannes Gross, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order". en. 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.

[17] Yuval Ishai, Amit Sahai, and David Wagner. "Private Circuits: Securing Hardware against Probing Attacks". en. In: *Advances in Cryptology - CRYPTO 2003*. Ed. by Dan Boneh. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, pp. 463–481. ISBN: 978-3-540-45146-4. DOI: 10.1007/978-3-540-45146-4_27.

[18] Arpan Jati et al. *SPQCop: Side-Channel Protected Post-Quantum Cryptoprocessor*. Cryptology ePrint Archive 2019/765. June 2019.

[19] Çetin K. Koç, ed. *Cryptographic Engineering*. en. New York, NY, USA: Springer, 2009. ISBN: 978-0-387-71816-3 978-0-387-71817-0.

[20] Stefan Mangard and Kai Schramm. "Pinpointing the Side-Channel Leakage of Masked AES Hardware Implementations Notes in Computer Science". en. In: *CHES*. 2006, p. 474.

[21] Stefan Mangard et al. "Side-Channel Leakage of Masked CMOS Gates". In: *Topics in Cryptology – CT-RSA 2005*. Ed. by Moni Naor et al. Vol. 3376. LNCS. Berlin, Heidelberg: Springer, 2005, pp. 351–365. ISBN: 978-3-540-24399-1 978-3-540-30574-3. DOI: 10.1007/978-3-540-30574-3_24.

[22] Vincent Migliore et al. "Masking Dilithium: Efficient Implementation and Side-Channel Evaluation". en. In: *Applied Cryptography and Network Security*. Ed. by Robert H. Deng et al. Vol. 11464. Cham: Springer International Publishing, 2019, pp. 344–362. ISBN: 978-3-030-21567-5 978-3-030-21568-2. DOI: 10.1007/978-3-030-21568-2_17.

[23] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. "Threshold Implementations Against Side-Channel Attacks and Glitches". In: *Information and Communications Security, ICICS 2006*. Vol. 4307. LNCS. Springer Berlin Heidelberg, 2006, pp. 529–545. ISBN: 978-3-540-49496-6 978-3-540-49497-3. DOI: 10.1007/11935308_38.

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

[25] Oscar Reparaz. "Analysis and Design of Masking Schemes for Secure Cryptographic Implementations". Ph.D. Thesis. Leuven, Belgium: KU Leuven, June 2016.

[26] Oscar Reparaz et al. *A Masked Ring-LWE Implementation*. Tech. rep. 724. 2015.

[27] Tobias Schneider, Amir Moradi, and Tim Güneysu. "Arithmetic Addition over Boolean Masking". en. In: *Applied Cryptography and Network Security*. Ed. by Tal Malkin et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 559–578. ISBN: 978-3-319-28166-7. DOI: 10.1007/978-3-319-28166-7_27.

[28] Tobias Schneider et al. "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.

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

[30] Kris Tiri, Moonmoon Akmal, and Ingrid Verbauwhede. "A Dynamic and Differential CMOS Logic with Signal Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards". en. In: *ESSCIRC*. 2002, pp. 403–406.

[31] Verhulst, Kasper. "Power Analysis and Masking of Saber". Master of Science in Mathematical Engineering. KU Leuven, 2019.

[32] Yihong Zhu et al. *A High-Performance Hardware Implementation of Saber Based on Karatsuba Algorithm*. Cryptology ePrint Archive 2020/1037. Aug. 2020.