SABER-GPU: A Response-Based Cryptography Algorithm for SABER on the GPU

Kaitlyn Lee, Michael Gowanlock, Bertrand Cambou
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, U.S.A.
kdl222@nau.edu, Michael.Gowanlock@nau.edu, Bertrand.Cambou@nau.edu

***Abstract*—The Internet of Things (IoTs) contain many low-powered devices that require secure communication. Post-quantum cryptography (PQC) is needed to address quantum computers that eventually will be able to break current encryption schemes. However, IoT devices are low-powered and often do not have the computational power to carry out computationally expensive error correction schemes. To address this limitation, we propose leveraging response-based cryptography (RBC) to secure IoT devices. Physical unclonable functions (PUFs) can replace random number generators that are used in the public key creation procedures. In this scheme, a server authenticates a client by generating the same seed from the image of the client's PUF stored in the server in a secure environment, thus replacing static public keys with dynamically generated key pairs. In this paper, we focus on generating the public key from the SABER PQC algorithm.**

Due to the inherent bit error rates existing in PUF technology, the protocol requires that the server recognizes the erratic keys. This error correction requires a massive parallel search over a key space bounded by the expected PUF error rate. This paper examines the use of parallel computing technologies to rapidly find a client's public key within reasonable time constraints. In particular, we examine using multi-core CPUs and many-core Graphics Processing Units (GPUs) in shared-memory environments. The design space for the SABER PQC algorithm is large. Therefore, we focus on performance engineering several CUDA kernels used in the RBC search that systematically explore this space. Our RBC search algorithms are highly scalable: the multi-core CPU algorithm achieves a speedup of 61.82× on 64 CPU cores, and our multi-GPU algorithm achieves a near-perfect speedup of 2.93× on 3 GPUs. Using a typical PUF error rate that requires searching 1.75×10^8 keys, we find that our GPU algorithm can authenticate a user within 6 seconds, which is well below our authentication time threshold.

***Index Terms*—General Purpose Computing on Graphics Processing Units, Parallel Computing, Physical Unclonable Functions, Post-Quantum Cryptography, Response-based Cryptography, SABER**

I. Introduction

With the rise of quantum computers, the use of canonical, cryptographic algorithms such as Rivest–Shamir–Adleman (RSA) [1] and Elliptic Curve Cryptography (ECC) [2] will no longer be secure [3]. Consequently, post-quantum cryptoraphic (PQC) algorithms have been proposed that are quantum-resistant. Currently, the United States' National Institute of Standards and Technology (NIST) is leading the effort to standardize the next-generation of cryptographic systems that are quantum-resistant. NIST is currently on its third round with 7 contenders in two categories: key encapsulation mechanisms (KEMs) and digital signature algorithms (DSAs). The four KEMs are Classic McEliece [4], CRYSTALS-Kyber [5], NTRU [6], and SABER [7], and the three DSAs are CRYSTALS-Dilithium [8], FALCON [9], and Rainbow [10].

Despite new quantum-resistant algorithms, mainstream public key infrastructure (PKI) requires that clients store private keys in memory (e.g., RAM, disk, etc.) on their devices, which can be recovered by an attacker. For low-powered IoT devices (e.g., sensor networks), storing private keys in this manner implies that an attacker may be able to physically access the device and read the private keys. To mitigate this concern, the use of physical unclonable functions (PUFs) can be used to generate keys on-demand without storing the keys in memory. This hardware acts as a digital "fingerprint" and is unique due to variations in production [11]–[13]. Additionally, the output of a PUF may change over time due to aging and environmental factors, causing the PUF to produce keys with large bit error rates. This means that a party's public/private key pair generation will be subjected to uncertainties.

To enable one-time keys and other cryptographic schemes that allow clients to change their public/private keys over time, certificate authorities (CA) will need to validate these client public keys. Response-based cryptography (RBC) can be employed in the CA scheme to authenticate client public keys. This protocol authenticates clients equipped with PUFs that generate a public/private key pair [14]. Since low-powered IoT devices may be unable to perform error correction and helper functions [14] due to the large computational and power costs, RBC requires that a high-powered server performs the correction to the output stream of the PUF. This assumes an asymmetric relationship between the computational capabilities between the client and server, where the latter is located in a secure environment. To determine whether a public key is valid, the server will conduct a search process, using PUF enrollment information stored on the server. This will benefit from a parallel search over a large key space.

This paper examines the use of new parallel architectures to carry out the search for a client's public key using the SABER post-quantum algorithm. In particular, we optimize the RBC search process on Graphics Processing Unit (GPU) hardware. In addition, we compare our optimized GPU algorithm to a multi-core CPU algorithm. We show that superior performance and scalability can be obtained through our GPU algorithm over the CPU algorithm. In summary, this paper makes the following major contributions:

- We identify the key components of SABER that are needed in the RBC protocol and are major targets for optimization. To this end, we propose several optimizations to the SABER key generation methods and to the RBC scheme for GPU architectures. To our knowledge, ours is the first SABER algorithm to be developed for the GPU.
- We compare the GPU algorithm to a shared-memory, multi-core CPU algorithm parallelized using OpenMP on up to 64 cores. We find that our multi-core algorithm is highly scalable in this environment.
- We also examine early-exit strategies for each algorithm to efficiently terminate the search when the key is found. We find that the early-exit procedure leads to minor performance degradation relative to the exhaustive search method.

The paper is organized as follows: Section II reviews some key information and presents related work, Section III presents our two algorithms, Section IV describes our two modes of execution, Section V introduces our optimizations made to SABER and RBC, Section VI presents our experimental evaluation, and Section VII summarizes our work.

II. Background

In this section, we describe SABER, the RBC protocol, high performance computing, and related works of literature.

A. SABER

The SABER KEM can be used to securely share symmetric keys between two parties. For completeness, we describe the steps for a KEM below:

1) Party 1 sends their public key, P_1, to party 2.
2) Party 2 randomly generates a symmetric key S (also called a session key), and encapsulates it in a ciphertext, C, using P_1.
3) Party 2 sends C to party 1.
4) Party 1 uses its secret key to decapsulate C to obtain S.

Once S is in the possession of both parties, party 1 and 2 may communicate securely using a symmetric-key encryption algorithm. In the RBC search, SABER is used for authentication and not for key exchange. For this reason, we focus on the SABER key generation functions.

The three main components of SABER are as follows: learning with rounding (LWR), polynomial multiplication, and hashing. For clarity we use the notation used in the SABER specification in our descriptions below [15].

1) Learning with Rounding: SABER is a set of lattice-based algorithms using the LWR problem. LWR, first introduced by Banerjee [16], is a variant of the learning with errors (LWE) problem, first introduced by Regev [17]. The LWE problem states that given a known matrix A, a random secret vector y, a small, error vector e, and a public vector $z = A \cdot y + e$, it is nearly impossible to distinguish y and e given A and z. Instead of adding a small, error vector e, LWR deterministically solves for $z = round(A \cdot y)$ by using a rounding function. Because we do not need e, this eliminates noise sampling and decreases the bandwidth of the problem [15], [18]. SABER uses two power-of-2 moduli p and q in its computation, where $q = 2^{13}$ and $p = 2^{10}$. During the rounding step, SABER rounds elements in a mod q space to a mod p space [7]. Because p and q are powers-of-2, modular reduction can be replaced by bit shift operations [15]. SABER then builds upon the LWR problem with Mod-LWR by defining y and z as matrices instead of vectors.

2) Polynomial Multiplication: One downfall of using power-of-2 moduli is the incompatibility with number theoretic transforms (NTT) [19]. Instead, SABER uses divide-and-conquer 4-way Toom-Cook [20], [21], Karatsuba [22], and schoolbook multiplication. Toom-Cook and Karatsuba use a top-down recursive approach by breaking the polynomials into smaller pieces. Then, schoolbook multiplication is used for coefficient multiplication. The authors of SABER use this approach because it decreases the total number of multiplication operations compared to using only schoolbook multiplication. [7].

3) Extendable Output Function and Hashing: SABER uses extendable output functions (XOF) to expand seeds to matrices. The XOF that SABER uses is `SHAKE128`. Additionally, SABER uses `SHA3_256` and `SHA3_512` for hashing. All three functions were introduced by Dworkin [23].

4) SABER Key Generation: In this paper, we focus on the key generation functions and do not discuss the encryption, decryption, encapsulate, and decapsulate functions of SABER because we do not use these functions for RBC. The SABER key generation function is described below:

1) Randomly generate the 256-bit seeds a and y using a random number generator (RNG),
2) create matrix A from seed a and matrix Y from seed y using `SHAKE128`,
3) calculate $z = round(A \cdot Y)$,
4) define the public key as (z, A),
5) hash the public key using: $pkh =$ `SHA3_256`(pk),
6) randomly generate the 256-bit seed c,
7) and define the private key as (Y, pk, pkh, c).

We now describe how SABER is incorporated into the RBC protocol.

B. Response-Based Cryptography (RBC)

In this section, we describe the process of RBC. Before authentication can take place, the server, located in a secure environment, needs to gather information about the client's PUF in an enrollment phase. During enrollment, the client will generate initial responses from its PUF, responding to challenges sent by the server. The server stores these challenge-response pairs in a lookup table with the index of the entry being computed with the user's ID uid. The steps for authentication, outlined in Figure 1, are as follows:

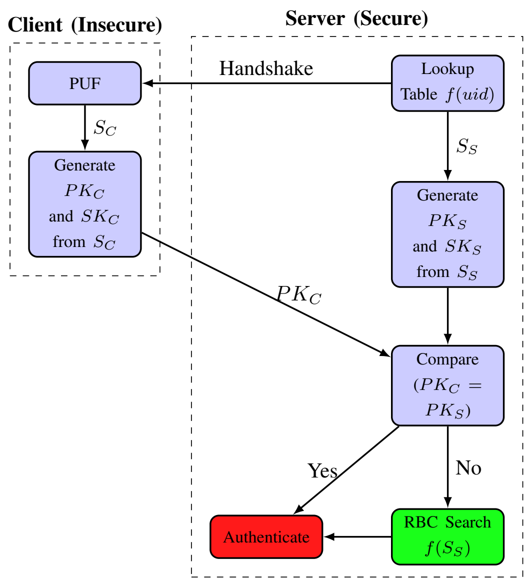

Fig. 1. Illustration of response-based cryptography with SABER. S_C - client's PUF seed, PK_C - client's public key, SK_C - client's private key, uid - client's user ID, S_S - server's PUF seed, PK_S - server's public key, and SK_S - server's private key.

1) The server will send instructions to the client via a handshake.
2) The client will use the instructions to generate responses from its PUF.
3) The seed S_C generated from the client's PUF is used to generate a public and secret key for the client, PK_C and SK_C.
4) The client sends PK_C to the server.
5) The server uses the client's initial responses from the client's PUF stored in the lookup table to generate the initial seed, S_S.
6) The server generates a public and private key, PK_S and SK_S, from S_S.
7) The server compares the public key sent from the client and the one it just generated,
8) if the public keys match, the client is authenticated.
9) If the keys do not match, the server initiates the RBC search on S_S.

The RBC search algorithm is described below:

1) Starting at a Hamming distance $d = 1$, the server will flip one bit in S_S and continue the authentication process starting at step 6 above.
2) Once all seeds one Hamming distance away have been searched, the server will start searching over seeds with a Hamming distance of two, and so on.
3) The search stops once the client's public key is found or when a maximum timing threshold, T, is met.

Due to the inherent noise of the PUF, the challenge-response pairs stored in the server's lookup table may be different from the responses generated by the client's PUF using the same challenges, causing S_C and S_S to be different at the start of the authentication process. In this paper, we define S_S and S_C to be 256 bits and define the number of seeds needed to be searched as a function h of d as follows:

$$h(d) = \sum_{i=1}^{d} \binom{256}{i} \tag{1}$$

For example, when the RBC search begins at $d = 1$, the server needs to iterate over $h(1) = \binom{256}{1} = 256$ seeds. If the search reaches a Hamming distance of 5, the server will need to iterate over $h(5) = \binom{256}{1} + \binom{256}{2} + \binom{256}{3} + \binom{256}{4} + \binom{256}{5}$ $\approx 8.9 \times 10^9$ seeds. In order to make the protocol responsive to the client, a timing threshold is set such that if the client's key is not found before the timing threshold, the server will ask the client to generate a new seed. For the purposes of this paper, we define the timing threshold as $T = 20$ seconds.

Cambou et al. [14] proposed an RBC scheme using the Advanced Encryption Standard (AES) and showed that it would take a single-core machine 10 hours to search up to $d = 5$. Using SABER instead of AES, we expect the RBC search to take longer because the public/private keys are longer and take longer to generate for SABER. Although this may seem like it would negatively impact the RBC protocol, the longer key-generation function makes it harder for an opponent to find the PUF seed within a given time constraint. Thus, the expensive key generation procedure is a security benefit. However, because of the large search space, using high-performance computing techniques is necessary to authenticate a user.

The complexity of the RBC search is also a function of the PUF error rate. To ensure that PUFs have an acceptable level of error, we employ the Ternary Addressable Public Key Infrastructure (TAPKI) protocol [24]. In short, the cells of the PUF are examined to determine which cells are stable with high probability, indicating that the bit is unlikely to be flipped when cycling the power. Those cells that are unstable are masked such that the addresses are excluded when performing the RBC search.

Threat Model: For clarity, we outline the security assumptions made by RBC. First, the server that performs the search is located in a secure environment such as that owned by a government entity or financial institution. Second, the PUFs are manufactured in a trusted environment and are also enrolled at this stage and stored on the server in this environment. Third, once the PUF is deployed on a client device for use, it is no longer considered to be secure.

C. High Performance Computing Overview

In this paper, we propose algorithms that exploit multi-core CPUs using OpenMP [25] and many-core GPUs containing several thousand cores using CUDA [26].

OpenMP is a library for C/C++ that uses shared-memory and CPU cores to compute work. Because OpenMP runs on a shared memory system, each thread can access memory shared between all threads.

CUDA is a programming interface in C/C++ that uses shared-memory and GPU threads. Threads are grouped together to form a block and blocks are grouped to form a grid.

Groups of 32 threads define a warp, which is a hardware constraint, where warps get assigned for execution on streaming multiprocessors (SMs). A GPU program is executed using a kernel function that is callable by the host (CPU) to be run on the device (GPU). When calling the kernel function, the programmer must provide the number of threads per block, b, and the total number of blocks to run the program with. Because warps contain 32 threads, it is best practice to run a multiple of 32 threads in a block.

We employ these parallel computing technologies to assess the performance of the RBC search using various target platforms.

D. Related Work

In this section, we review the literature exploring SABER, PUFs, and RBC.

Cambou et al. [27] showed how the NIST PQC candidates fit into the RBC scheme using PUFs. Although the authors describe RBC with SABER, their work is limited to preliminary results using a multi-core CPU. In contrast, this paper explores optimizing SABER for the RBC search, which is conducted using multi-core CPUs and many-core GPUs.

There are currently many works that optimize SABER. Mera et al. [28] optimize polynomial multiplication, reducing the algorithm's execution time. In particular, they propose faster Toom-Cook multiplication using lazy interpolation. Although this work speeds up polynomial multiplication in SABER on the CPU, it increases the memory footprint. Initial experiments showed that this optimization on the GPU degrades performance, so we do not report on this method in our evaluation.

Zhu et al. [29] also focus on optimizing polynomial multiplication on FPGA hardware by using 8-way Karatsuba multiplication, reducing the total number of multiplication operations from 65,536 to 6,521. They also propose a scheduling approach for Karatsuba that reduces register usage on FPGA hardware. SaberX4, proposed by Roy [30], uses AVX2 instructions to parallelize 4 key generation calls, yielding an increase in key generation throughput of 38% compared to the non-vectorized baseline SABER. While the last two works use AVX2 CPU instructions and FPGA hardware, these optimizations are unsuitable for the GPU. However, the focus of our work on optimizing SABER on the GPU is similar to these works as we similarly consider hardware constraints when designing our optimizations.

III. Response-Based Cryptography Algorithm on the GPU

We propose two different SABER RBC algorithms using OpenMP and CUDA, denoted as SABER-OpenMP and SABER-GPU, respectively. SABER-OpenMP makes use of CPU threads to accelerate the RBC search and SABER-GPU uses GPU threads. We only describe SABER-GPU below due to space constraints; however, both algorithms are similar.

Algorithm 1 highlights the integration of baseline SABER with RBC. In this paper, since we focus on carrying out

Algorithm 1 RBC Search Algorithm

```
1: procedure RBCSearch($d$)
2:     ($S_C$, $a$, $PK_C$, $SK_C$) ← generateClientData()
3:     $S_S$ ← corruptClientSeed($S_C, d$)
4:     $authSeed$ ← $\emptyset$
5:     $found\_key\_flag$ ← false
6:     if $S_S == S_C$ then
7:         return True
8:     else
9:         for $i \in 1,\ldots,d$ do
10:            $k$ ← totalNumKeys($d$)
11:            $t$ ← totalNumThreads($k$)
12:            $n$ ←KeysPerThread($k, t$)
13:            rbcSearchKernel($authSeed$, $S_S$, $PK_C$, $a$, $i$, $n$, $found\_key\_flag$)
14:        return ($authSeed$ = $S_C$)
15:
16: procedure RBCSearchKernel($authSeed$, $S_S$, $PK_C$, $a$, $i$, $n$, $found\_key\_flag$)
17:     $tid$ ← getThreadId()
18:     ($startComb, endComb$) ← getCombPair($tid, i, n$)
19:     $seedIter$ ← getIterator($S_S, startComb, endComb$)
20:     while !seedIter.end() and $found\_key\_flag$ is false do
21:         $PK_S$ ← createPublicKey($seedIter$.currSeed(), $a$)
22:         if $PK_S == PK_C$ then
23:             $authSeed$ ← $seedIter$.currSeed()
24:             $found\_key\_flag$ ← true
25:         $seedIter$.next()
26:     return
```

the RBC search phase in parallel, we use software to generate PUF bit streams instead of using PUFs. The procedure `RBCSearch` is ran on the host and starts by generating client data on line 2, consisting of the private seed S_C, a random public seed a that is generated from a RNG, and the client's public and private keys PK_C and PK_S. Line 3 corrupts S_C by flipping d bits. This corrupted seed, S_S, is the first seed that will be searched by the server. We create an empty $authSeed$ on line 4 that will store the seed that generates the matching public key. We also create $found\_key\_flag$ on line 5 and set it to false. Line 6 checks if S_S and S_C are the same, and if they are, we return $True$ on line 7. If the seeds are not the same, we iterate through each Hamming distance from 1 to d on line 9. For each Hamming distance, we calculate the total number of keys k on line 10, the total number of threads t on line 11, and the number of keys searched by each thread n on line 12. Finally, line 13 calls the `rbcSearchKernel` executed on the GPU, where the search occurs at a Hamming distance of i.

Algorithm 1 also describes the GPU kernel, `rbcSearchKernel`, that parallelizes the search. The kernel takes the empty authentication seed $authSeed$, the server's starting seed S_S, the client's public key PK_C, seed a, the current Hamming distance i to search over, the number of keys per thread n, and the flag used to notify all GPU threads that the key has been found, $found\_key\_flag$, as input.

The kernel starts with each GPU thread generating a unique ID tid on line 17. In order for each thread to independently search the key space without any coordination, reducing overhead over other methods that require coordination between threads, line 18 generates a starting and ending combination pair for that thread using tid, i, and n. These combination pairs are used to create a seed iterator that will iterate through the key space starting at the starting combination and ending at the ending combination on line 19. Next, we iterate through the the thread's key space, checking if the key has been found on line 20. In this loop, line 21 generates PK_S using the current seed and a. Line 22 checks if the keys match, and if they do, set $authSeed$ to the current seed on line 23 and set $found\_key\_flag$ to true on line 24. If the keys do not match, generate the next seed on line 25.

SABER-OpenMP and SABER-GPU follow Algorithm 1, with a few minor differences due to the fact that they are ran on different architectures. However, each algorithm has a different early-exit strategy, described in the next section.

IV. Early-Exit Strategies for Response-Based Cryptography

In our evaluation, we consider two modes of the RBC search. The first mode is an exhaustive search where we assume that all keys up to and including a Hamming distance of d need to be searched. The second mode is where the algorithm will stop searching early once the matching seed has been found, and a thread will broadcast this information to the other threads such that all threads terminate their searches early. While the exhaustive search is straightforward, we outline the early-exit procedures for each algorithm below. It is important to note that both algorithms make use of a variable called $found\_key\_flag$ to know if the key was found.

SABER-OpenMP: Because each CPU thread has its own private memory, the threads need a way to communicate that the key was found. We achieve this by storing $found\_key\_flag$ in shared memory that is accessible to all threads. When one of the threads finds the key, it will set $found\_key\_flag$ to true. Then, before each thread generates another key, it will check the value of the flag and stop its search if it is true.

SABER-GPU: We store $found\_key\_flag$ in unified memory that is accessible to the host and device. When a thread finds the key, it will set the flag. Then, when each thread checks the value of $found\_key\_flag$, the search will stop.

V. SABER-GPU Optimizations

In this section, we describe several optimizations to improve the performance of the RBC search process. Optimizations described in Sections V-B, the first part of V-C, V-D, and V-E are included in both SABER-OpenMP and SABER-GPU. While optimizations described in Sections V-C (the second part), V-F, and V-G are specific to SABER-GPU. These optimizations were not shown in Algorithm 1.

A. Public Key Size

There are three different security levels of SABER: LightSABER, SABER, and FireSABER. The sizes of the public keys for the different security levels are 672, 992, and 1312 bytes for LightSABER, SABER, and FireSABER, respectively [7]. Because of the time constraint to authenticate a user and the fact that the time to authenticate depends on the size of the public keys, we chose to use the lowest security level, LightSABER. This reduces the amount of bits we need to compare during the RBC search, thus reducing the total execution time. We do not discuss the other two security levels in this paper.

B. Splitting the Key Generation

The SABER key generation function generates both the public and private keys; however, only client and server public keys are needed in our algorithm. Because of this, we remove the private key generation from the RBC algorithm. In return, this decreases memory usage and computation because the private key size for LightSABER is 832 bytes and we remove a single call to the `SHAKE128` subroutine [15].

C. Optimizing the Storage and Generation of Matrix A and SHA Constants

Removing Matrix A Regeneration: The architecture of the baseline SABER algorithm uses seed a in `createPublicKey` to generate matrix A (Section II-A4). This means that each GPU thread will be generating the same A. Instead of passing a to the kernel and computing A on the GPU, we generate A on the host and pass it as an argument to `rbcSearchKernel`, thus preventing regenerating A on the GPU. We then modify the public key generation method by removing the matrix A generation and instead use the matrix A that is stored in the GPU's global memory.

Shared Memory: Off-chip global memory has higher memory access latency than on-chip shared memory [26]. Thus, shared memory can be used as a temporary location to cache data to be reused multiple times by blocks of threads. In addition to removing A, as described above, each block copies A from global memory to shared memory to exploit this faster storage location. Since A is 1,024 elements [15], this reduces a significant number of transactions to global memory.

Similarly, we also store the 24 hashing constants used by `SHA3` in shared memory.

D. Using One-Dimensional Arrays

SABER uses 2D and 3D arrays to store its matrices. Instead, we "flatten" these 2D and 3D arrays to 1D arrays. This increases spatial locality since we only need to allocate a single, contiguous memory location to store the matrices.

E. Refactor Polynomial Multiplication

We profiled SABER-OpenMP, instead of SABER-GPU because the CUDA profiler has limited functionality, to determine the time spent in the major functions of the algorithm. We found that the majority of the execution time is spent performing polynomial multiplication and extending seeds to matrices using `SHAKE128`. Below we summarize the fraction of time spent in `karatsuba_simple`, `toom_cook_4way`, `SHAKE128`, State_Permute, a method used in SHAKE128, and all other methods categorized in "Other":

- karatsuba_simple: 53.31%
- toom_cook_4way: 21.53%
- Other: 11.86%
- State_Permute: 11.63%
- SHAKE128: 1.67%

Based on the profiled operations described above, we examine improving the performance of the polynomial multiplication. We did this in two ways: removing temporary variables and moving the modular reduction step that occurs after the multiplication.

First, throughout karatsuba_simple and toom_cook_4way, there are variables used to store temporary results. In order to reduce the memory usage of these functions, we removed these temporary variables and instead store results directly in the result matrices.

Second, matrix A is of size $2 \times 2 \times 256$. Because of this, polynomial multiplication occurs in three nested loops. The modular reduction step that reduces the entries of the output matrix by the mod $q = 2^{13}$ and originally occurred in the third inner loop. Polynomial multiplication is used to multiply matrix A, a 3D matrix, and the secret matrix Y, a 2D matrix of size 2×256, resulting in a 2D matrix of size 2×256. Because of this, the modular reduction step can be moved out of the third loop and into the second, reducing the number of reduction calls by a factor of 2.

F. Keys Per Thread and Threads Per Block

Keys per thread (n): Each GPU thread is assigned to a single SM, where each SM contains multiple cores. There is overhead in launching GPU kernels as a function of the number of threads that are created. Each thread can be assigned one or more keys to process. With many threads, the resources will be saturated at the potential expense of more thread creation overhead. Thus, there is a trade-off between thread creation overhead and GPU utilization. We experimentally evaluate the best number of keys that are assigned to each thread such that we reach a good trade-off between overhead and GPU resource utilization.

Threads per block (b): A CUDA block of threads is assigned to a single SM. Each SM has resources, such as registers and shared memory. The number of threads per block will impact performance, as some thread block sizes may potentially leave resources underutilized on the SM. Consequently, we experimentally determine a good block size that yields the best performance. Since groups of 32 threads execute on a SM, we examine values that are divisible by this size.

G. Using AES CTR

As previously stated, the second bottleneck of the SABER key generation function is SHAKE128, including its child function State_Permute. SABER uses SHAKE128 as an XOF when creating matrix A and the secret matrix Y. The authors of SABER recommend replacing SHAKE128 with AES in counter (CTR) mode [15]. They demonstrate that using AES results in a reduction of cycle counts from 66,727 to 36,315 in their AVX2-optimized implementation. We replicate this work on the GPU.

TABLE I
SUMMARY OF VARIABLES DISCUSSED IN PREVIOUS SECTIONS.

Variable	Definition
S_S	Server's starting PUF seed.
A	Public matrix used for key generation.
n	Keys searched per thread.
b	Threads per block.
d	Hamming distance client seed is corrupted by.

H. Public Keys in Global Memory

When profiling SABER-GPU, we found that the maximum number of registers were being used per thread (255). Using this maximum decreases the total number of threads that can be executed at any given time, thus degrading performance. To attempt to reduce register usage, we store the server-generated public keys, PK_S, in global memory instead of in registers.

I. Public Keys in Shared Memory

Recall from Section V-C, global memory has high memory access latency, while shared memory has lower latency. For this reason, we examine storing the public keys in shared memory. Because of the small size of shared memory and the large size of the public keys, we limit b to 32.

VI. Experimental Evaluation

In this section, we present our performance evaluation. For clarity, Table I summarizes the variables used throughout the evaluation.

A. Experimental Methodology

During our experiments, we use one platform equipped with 3×Tesla A100 GPUs, each with 6912 GPU cores and 40 GiB of memory, and 2 AMD EPYC 7542 CPUs, each with 32 cores, running at a clock speed of 2.9 GHz and sharing 512 GiB of memory.

All CPU code is written in C/C++ and compiled with the O3 compiler flag. All GPU code is written in CUDA and our platform uses CUDA 11. Our results are the average of five time trials and focus on the throughput of keys searched per second and the execution time. We gather two types of results using the two modes described in Section IV: an exhaustive search and the early-exit strategy. Regarding the early-exit strategy, we set the server starting seed, S_S, to the middle seed of the search space at Hamming distance d. This means that half of the seeds at a Hamming distance d are searched, while all Hamming distances $< d$ are exhaustively searched.

B. Summary of Algorithms and Configurations

Below, we define our two algorithms described in Section III and their configurations. We also define p as the number of CPU cores and g as the number of GPUs used while executing our algorithms.

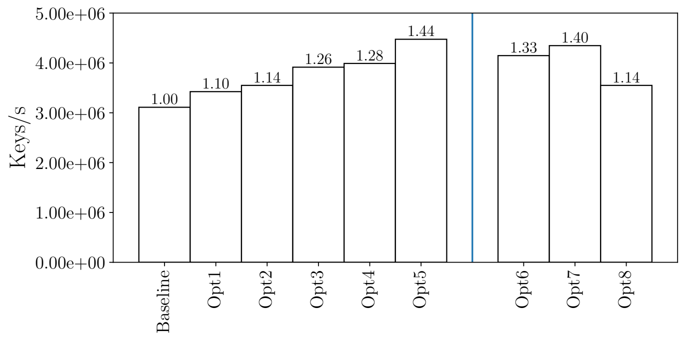

Fig. 2. SABER exhaustive key search throughput in keys/s at a Hamming distance of $d = 3$ for each optimization described in Section V. Above each bar is the increase in throughput over the baseline. The vertical line divides the optimizations that provide an increase in throughput (left) from those that do not after optimization set #5 (right). See Table II for optimization definitions.

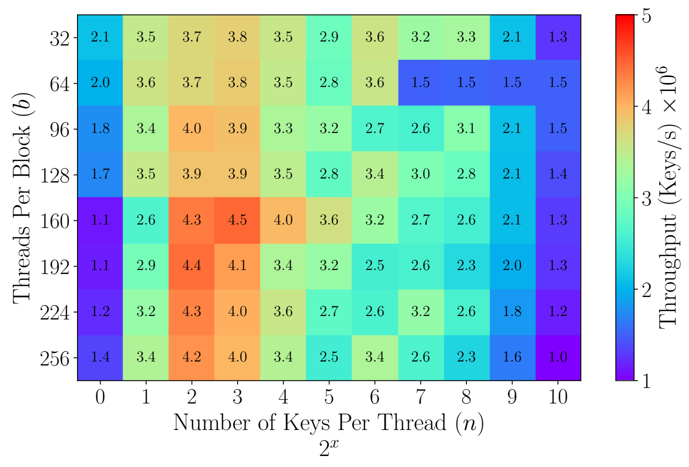

Fig. 3. Heat map showing the exhaustive search throughput in keys/s at a Hamming distance of $d = 3$ for different combinations of threads per block and number of keys per thread.

- SABER-OpenMP: our OpenMP implementation. Experiments are ran with $p = 1$ and $p = 64$ cores.
- SABER-GPU: our GPU implementation with $g = 1 - 3$ GPUs. From our experiments, we set $n = 8$ and $b = 160$. We will show that these parameters obtain the best performance.

C. Speedup and Parallel Efficiency

We will calculate the speedup and parallel efficiency for our two algorithms during our evaluation to get a better understanding of how they perform. We define speedup as the performance gained when running SABER-OpenMP with $p = 1$ vs. $p = 64$ and SABER-GPU with $g = 1$ vs. $g = 2 - 3$ and is calculated by:

$$\text{Speedup}(p \text{ or } g) = \frac{E_1}{E_m}, \tag{2}$$

where E_1 is the execution time for SABER-OpenMP and SABER-GPU when $p = 1$ and $g = 1$ and E_m is the execution time when $p = 64$ and $g = 2 - 3$, respectively. When the speedup is equal to p or g, we call this a perfect speedup. Parallel efficiency is defined as how well the resources are being used and is calculated by:

$$\text{Efficiency}(p \text{ or } g) = \frac{\text{Speedup}(p \text{ or } g)}{p \text{ or } g} \tag{3}$$

When a perfect speedup occurs, the parallel efficiency will be equal to 1.

TABLE II
Optimizations denoted in Figure 2 as Optimization sets #1–8. For reference, Section refers to location in the paper where the optimization was defined.

Optimization	Section	Optimization Set Number 1	2	3	4	5	6	7	8
Public Key Only	§ V-B	✓	✓	✓	✓	✓	✓	✓	✓
Shared Memory/Share A	§ V-C		✓	✓	✓	✓	✓	✓	✓
Use 1-D Matrices	§ V-D			✓	✓	✓	✓	✓	✓
Optimize Multiplication	§ V-E				✓	✓	✓	✓	✓
Optimize Threads	§ V-F					✓	✓	✓	✓
Use AES CTR	§ V-F						✓		
Keys in Global Memory	§ V-G							✓	
Keys in Shared Memory	§ V-H								✓

D. SABER-GPU: *Evaluation of Our Optimizations*

In Figure 2, we show the results of the optimizations described in Section V by plotting the SABER exhaustive key search throughput in keys/s at a Hamming distance of $d = 3$, starting with baseline SABER with no optimizations. We omit showing other values of d as results are similar. Each bar is labeled with the ratio of its throughput over the output of the baseline. In Table II, we show each optimization, the section where the optimization was described, and its corresponding optimization set number. Optimization sets #1–5 are cumulative, e.g., optimization set #3 contains the same optimizations as #1 and #2. Optimization sets #6–8 are disjoint, but contain optimization set #5.

Starting at baseline SABER, we observe that as we add optimizations up to optimization set #5, the throughput increases. Optimization set #5 produces 1.44× more keys than the baseline implementation. In contrast, optimization sets #6–8 did not perform as well as optimization set #5. In Figure 2, the vertical line separates the optimizations with an increased throughput from the ones that do not. Consequently, optimization sets #6–8 should not be used. Recall from Section V-G that optimization set #6 replaces `SHAKE128` with AES CTR. Using AES CTR instead may decrease throughput because it uses more memory than `SHAKE128`. Also, recall that optimization set #7 and #8, discussed in Section V-H and Section V-I, store the public keys generated by the server in global and shared memory, respectively. We believe that optimization set #7 was limited by global memory latency, and with optimization set #8, because of the small size of shared memory available on the GPU, we had to restrict b to 32, thus decreasing concurrency.

As previously mentioned in Section V-F, we optimized the number of keys searched by each thread, n, and the number of threads in a block, b. In order to find the best values for n and

TABLE III

PERFORMANCE OF THE EXHAUSTIVE RBC SEARCH USING A HAMMING DISTANCE OF $d = 4$. COLUMNS: ALGORITHM, NUMBER OF CPU CORES (p), NUMBER OF GPUS (g), EXECUTION TIME (S), SPEEDUP, AND PARALLEL EFFICIENCY. THE TWO ALGORITHMS ARE SEPARATED BY A HORIZONTAL LINE.

Algorithm	p	g	Time (s)	Speedup	Parallel Eff.
SABER-OPENMP	1	-	2755.85	1.0	1.0
SABER-OPENMP	64	-	44.58	61.82	0.97
SABER-GPU	-	1	41.24	1.0	1.0
SABER-GPU	-	2	20.83	1.98	0.99
SABER-GPU	-	3	14.03	2.93	0.98

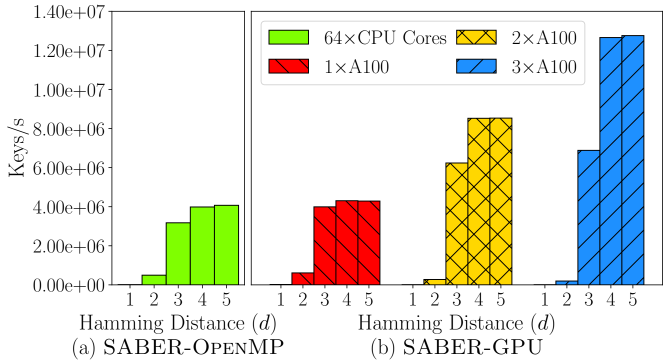

Fig. 4. Throughput in keys per second for an exhaustive search at Hamming distances $d = 1 - 5$. (a) SABER-OPENMP with $p = 64$ cores and (b) SABER-GPU with $g = 1 - 3$ GPUs.

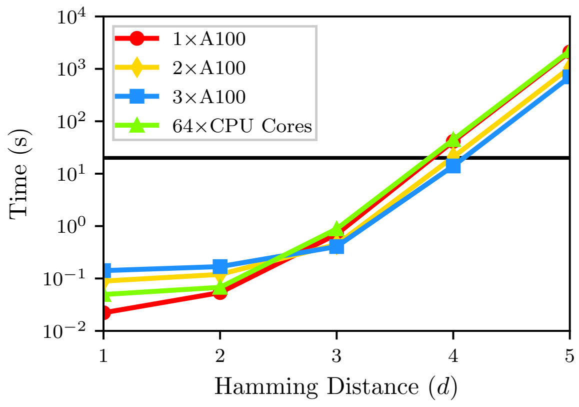

Fig. 5. Execution time in seconds for an exhaustive search at Hamming distances $d = 1 - 5$. We show the results for SABER-GPU using 1×A100 (circle markers), 2×A100 (diamond markers), and 3×A100 (square markers), and the results for SABER-OPENMP using 64 cores (triangle markers). The horizontal black line represents our timing threshold $T = 20$ seconds.

b, we evaluated multiple combinations of the two variables, running our experiments with optimization set #5. Figure 3 shows a heat map with b and n at a Hamming distance of $d = 3$. We omit showing other values of d as results are similar. As seen in the figure, setting $b = 160$ and $n = 2^3 = 8$ results in the highest key throughput. For this reason, we configure SABER-GPU with these values of b and n (as described in Section VI-B).

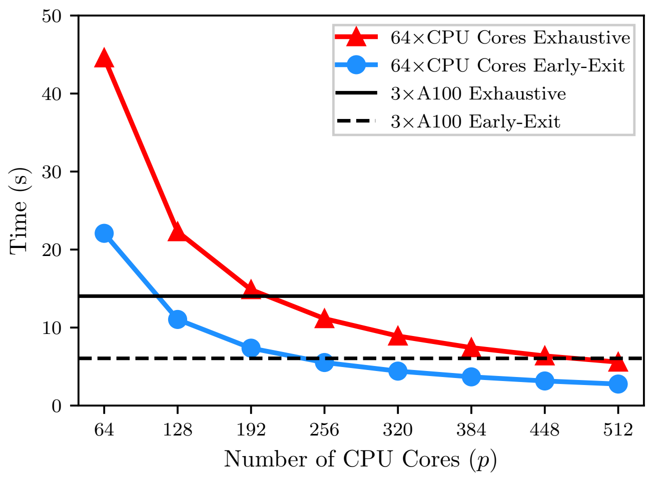

Fig. 6. Execution time in seconds at a Hamming distance of $d = 4$ vs. increasing number of cores p for SABER-OPENMP. We extrapolate our $p = 64$ results to calculate the execution time for increasing p values for both an exhaustive search (triangle markers) and the early-exit procedure (circle markers).The solid line indicates the execution time for SABER-GPU with $g = 3$ for an exhaustive search, while the dashed line indicates the execution time for SABER-GPU with $g = 3$ for the early-exit procedure.

E. Comparison of Algorithms: Exhaustive Search Results

Figure 4(a)–(b), shows the throughput in keys searched per second for an exhaustive search for SABER-OPENMP and SABER-GPU, respectively. We see that as d increases, the throughput increases because of the increase in search space; however, we do observe the throughput converging at $d \geq 4$. Figure 5 shows the execution time in seconds for SABER-OPENMP with $p = 64$ and SABER-GPU with $g = 1-3$. The execution times are plotted on a logarithmic scale because the time increases exponentially with increasing d. At $d = 1-2$, SABER-GPU with $g = 1$ performs the best compared to $g = 2-3$ and SABER-OPENMP with $p = 64$. This is due to the overhead of using $2-3$ GPUs on a small problem size. Additionally, SABER-GPU with $g = 1$ outperforms SABER-OPENMP with $p = 64$ for each d. Finally, SABER-GPU with $g = 3$ performs the best for $d \geq 3$. The black line in Figure 5 represents our timing threshold of $T = 20$ seconds. For $d \leq 3$, both algorithms can authenticate a client before $T = 20$ seconds with $p = 64$ and $g = 1-3$ for SABER-OPENMP and SABER-GPU, respectively; however at $d = 4$, only SABER-GPU with $g = 3$ is able to meet the authentication time limit. For $d \geq 5$, neither algorithms would be able to authenticate a client with our selected p and g values.

Table III shows the execution time, speedup, and parallel efficiency for both SABER-OPENMP and SABER-GPU. With $p = 64$ and $g = 3$ for SABER-OPENMP and SABER-GPU, we have parallel efficiencies of 0.97 and 0.98, respectively. These are near-perfect speedups and show that the SABER RBC search has great scalability on both the CPU and GPU.

In order to better compare SABER-OPENMP and SABER-GPU, Figure 6 plots the execution time at $d = 4$ for

TABLE IV
THE SAME AS TABLE III, BUT FOR AN EARLY-EXIT SEARCH.

Algorithm	p	g	Time (s)	Speedup	Parallel Eff.
SABER-OPENMP	1	-	1408.70	1.0	1.0
SABER-OPENMP	64	-	22.08	63.80	1.0
SABER-GPU	-	1	17.33	1.0	1.0
SABER-GPU	-	2	8.89	1.95	0.97
SABER-GPU	-	3	6.05	2.86	0.95

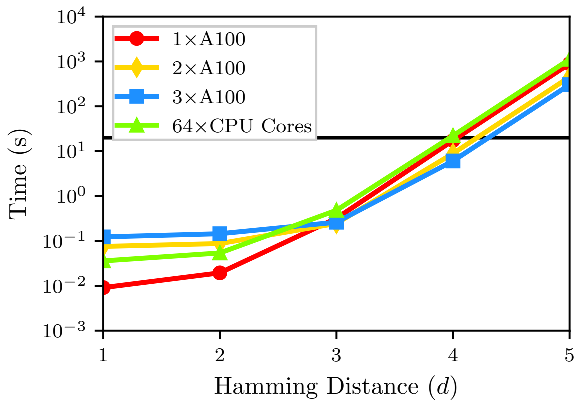

Fig. 7. The same as Figure 5, but for the early-exit search.

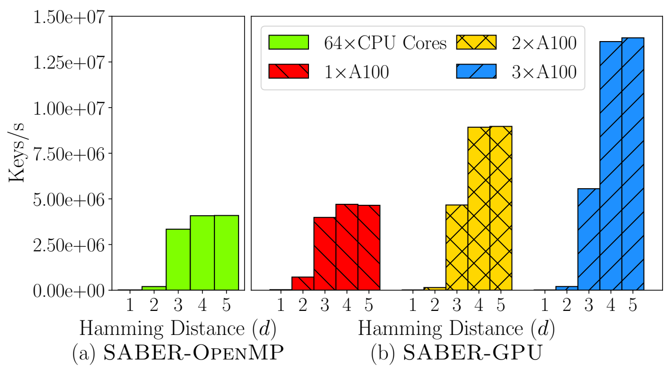

Fig. 8. The same as Figure 4, but for the early-exit search.

extrapolated values of p. This assumes perfect scalability and is therefore a lower-bound execution time estimate. Using the data from Figure 6, 3×A100 GPUs is equivalent to about 190 CPU cores for an exhaustive RBC search. However, scaling SABER-OPENMP to $p > 64$ requires using multiple compute nodes connected by a network, increasing inter-node communication latency. Additionally, this requires using a distributed-memory communication protocol that uses processes instead of threads, where processes have more overhead. On the other hand, in order to increase g, we would only need to add more GPUs to a single computer. This demonstrates that SABER-GPU obtains better scalability over SABER-OPENMP.

F. Comparison of Algorithms: Early-Exit Strategy Results

In this section, we discuss the results of the early-exit strategy using the same experimental scenarios as Section VI-E. Figure 7 shows the execution time for SABER-GPU with $g = 1-3$ and SABER-OPENMP with $p = 64$. Similar to Figure 5, SABER-GPU with $g = 1$ performs the best until $d = 3$. At $d \geq 3$, SABER-GPU with $g = 3$ outperforms the others. At $d = 4$, SABER-GPU with $g = 1-3$ can authenticate a user under the timing threshold $T = 20$, while SABER-OPENMP cannot with $p = 64$. Figure 8 shows the throughput in keys searched at varying hamming distances for SABER-OPENMP and SABER-GPU. We see the same convergence as Figure 4 at $d = 4-5$. Table IV shows the execution time, speedup, and parallel efficiency for SABER-OPENMP and SABER-GPU. We achieve near-perfect speedup for both SABER-OPENMP and SABER-GPU with $p = 64$ cores and $g = 2-3$ GPUs, respectively. However, we lose some efficiency for SABER-GPU using the early-exit procedure compared to the exhaustive search because of the need to communicate to each GPU that the key is found. We also see that we can authenticate a user at $d = 4$ in ≈6 seconds using $g = 3$ GPUs. The points plotted with circle markers in Figure 6 show the execution time for the early-exit procedure at varying p values. Using this data, we calculate that $p = 238$ cores is equivalent to $g = 3$ GPUs. Similarly to the exhaustive search, scaling SABER-OPENMP to $p = 238$ cores would increase execution time because of inter-node communication latency and the increased overhead of using processes. Therefore, SABER-GPU achieves better scalibility compared to SABER-OPENMP for the early-exit procedure.

VII. CONCLUSION

Near-future quantum computers will make standard cryptographic schemes obsolete. For this reason, NIST is facilitating a competition to select quantum secure cryptography algorithms. In this paper, we selected SABER, a KEM algorithm, that is a round three finalist in the competition.

To address the security of low-powered devices, such as those in the IoT, RBC allows these devices to authenticate with a secure server by validating the keys generated by client devices equipped with PUFs. Using PUFs allows for one-time keys and avoids storing private keys in non-volatile memory. To incorporate SABER into RBC, we optimized the public key generation procedures for the GPU. These optimizations achieved a speedup over the baseline implementation of 1.44×. We compared this optimized GPU algorithm to a parallel CPU algorithm, and found that a single GPU yields a key search throughput that is equivalent to 64 CPU cores. Using the GPU has several advantages over multi-core CPUs in terms of scalability. We can scale within a single computer using multiple GPUs. In contrast, scaling the CPU algorithm requires using multiple compute nodes, and this would degrade search performance due to inter-node communication. We also find that three GPUs can authenticate a client within 6 seconds using a Hamming distance of 4, which is an expected bit error rate for a PUF that has been configured using TAPKI.

Future work directions include comparing the GPU algorithm to a distributed-memory algorithm that is executed on a cluster of compute nodes, and investigating the GPU algorithm's high register usage in the SABER key generation procedure.

Acknowledgement

This material is based upon the work funded by the Information Directorate under AFRL award number FA8750-19-2-0503. Acknowledgment of support and disclaimer: (a) Contractor acknowledges Government's support in the publication of this paper. This material is partially based upon the work funded by the Information Directorate, under AFRL (b) Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFRL.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, p. 120–126, Feb. 1978.

[2] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology — CRYPTO '85 Proceedings*, H. C. Williams, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 417–426.

[3] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, p. 1484–1509, Oct 1997.

[4] B. Biswas and N. Sendrier, "McEliece Cryptosystem Implementation: Theory and Practice," in *Post-Quantum Cryptography*, J. Buchmann and J. Ding, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 47–62.

[5] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle, "CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM," in *2018 IEEE European Symposium on Security and Privacy*, 2018, pp. 353–367.

[6] A. Hülsing, J. Rijneveld, J. M. Schanck, and P. Schwabe, "High-speed key encapsulation from NTRU," Cryptology ePrint Archive, Report 2017/667, 2017.

[7] J.-P. D'Anvers, A. Karmakar, S. Sinha Roy, and F. Vercauteren, "Saber: Module-LWR Based Key Exchange, CPA-Secure Encryption and CCA-Secure KEM," in *Progress in Cryptology – AFRICACRYPT 2018*, A. Joux, A. Nitaj, and T. Rachidi, Eds. Cham: Springer International Publishing, 2018, pp. 282–305.

[8] "CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation. Part of the Round 3 Submission Package to NIST." accessed: April 20, 2021. [Online]. Available: https://pq-crystals.org/dilithium/

[9] "Falcon: Fast-Fourier Lattice-Based Compact Signatures over NTRU, Specification v1.2." accessed: April 20, 2021. [Online]. Available: https://pq-crys-tals.org/dilithium

[10] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, J. Kelsey, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone, "Status report on the second round of the nist post-quantum cryptography standardization process," 2020.

[11] I. Papakonstantinou and N. Sklavos, *"Physical Unclonable Functions (PUFs) Design Technologies: Advantages and Trade Offs"*. Springer International Publishing, 2018, pp. 427–442.

[12] C. Herder, M.-D. Yu, F. Koushanfar, and S. Devadas, "Physical unclonable functions and applications: A tutorial," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1126–1141, 2014.

[13] B. Cambou and M. Orlowski, "PUF Designed with Resistive RAM and Ternary States," in *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, ser. CISRC '16. New York, NY, USA: Association for Computing Machinery, 2016.

[14] B. Cambou, C. Philabaum, D. Booher, and D. A. Telesca, "Response-based cryptographic methods with ternary physical unclonable functions," in *Future of Information and Communication Conference*. Springer, 2019, pp. 781–800.

[15] "SABER: Mod-LWR based KEM (Round 3 Submission.)," accessed: April 20, 2021. [Online]. Available: https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/

[16] A. Banerjee, C. Peikert, and A. Rosen, "Pseudorandom functions and lattices," in *Advances in Cryptology – EUROCRYPT 2012*, D. Pointcheval and T. Johansson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 719–737.

[17] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *J. ACM*, vol. 56, no. 6, Sep. 2009.

[18] J. Alwen, S. Krenn, K. Pietrzak, and D. Wichs, "Learning with rounding, revisited," in *Advances in Cryptology – CRYPTO 2013*, R. Canetti and J. A. Garay, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 57–74.

[19] M. Bhattacharya, R. Creutzburg, and J. Astola, "Some historical notes on number theoretic transform," in *Proceedings of the 2004 International TICS Workshop on Spectral Methods and Multirate Signal Processing*, 2004.

[20] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," in *Soviet Mathematics Doklady*, vol. 3, no. 4, 1963, pp. 714–716.

[21] S. Cook and S. O. Aanderaa, "On the minimum computation time of functions," *Transactions of the American Mathematical Society*, vol. 142, pp. 291–314, 1969.

[22] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, no. 2. Russian Academy of Sciences, 1962, pp. 293–294.

[23] P. Pritzker and P. D. Gallagher, "Sha-3 standard: permutation-based hash and extendable-output functions," *Information Tech Laboratory National Institute of Standards and Technology*, pp. 1–35, 2014.

[24] B. Cambou, M. Gowanlock, J. Heynssens, S. Jain, C. Philabaum, D. Booher, I. Burke, J. Garrard, D. Telesca, and L. Njilla, "Securing additive manufacturing with blockchains and distributed physically unclonable functions," *Cryptography*, vol. 4, no. 2, p. 17, 2020.

[25] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[26] D. De Donno, A. Esposito, L. Tarricone, and L. Catarinucci, "Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD [EM Programmer's Notebook]," *IEEE Antennas and Propagation Magazine*, vol. 52, no. 3, pp. 116–122, 2010.

[27] B. Cambou, M. Gowanlock, B. Yildiz, D. Ghanaimiandoab, K. Lee, S. Nelson, C. Philabaum, A. Stenberg, and J. Wright, "Post quantum cryptographic keys generated with physical unclonable functions," *Applied Sciences*, vol. 11, no. 6, 2021.

[28] J. Mera, A. Karmakar, and I. Verbauwhede, "Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 222–244, 03 2020.

[29] Y. Zhu, M. Zhu, B. Yang, W. Zhu, C. Deng, C. Chen, S. Wei, and L. Liu, "A High-performance Hardware Implementation of Saber Based on Karatsuba Algorithm," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 1037, 2020.

[30] S. Sinha Roy, "SaberX4: High-Throughput Software Implementation of Saber Key Encapsulation Mechanism," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 2019, pp. 321–324.