Evaluating Accelerators for a High-Throughput Hash-Based Security Protocol

Kaitlyn Lee
Northern Arizona University
Flagstaff, AZ, USA
kdl222@nau.edu

Brian Donnelly
Northern Arizona University
Flagstaff, AZ, USA
bwd29@nau.edu

Tomer Sery
GSI Technology
Tel Aviv-Yafo, Israel
tsery@gsitechnology.com

Dan Ilan
GSI Technology
Tel Aviv-Yafo, Israel
dilan@gsitechnology.com

Bertrand Cambou
Northern Arizona University
Flagstaff, AZ, USA
bertrand.cambou@nau.edu

Michael Gowanlock
Northern Arizona University
Flagstaff, AZ, USA
michael.gowanlock@nau.edu

ABSTRACT

Security threats are rising due to widely available computational power and near-future quantum computers. New cryptographic protocols have been developed to address these challenges, but very few protocols take advantage of parallel computing. In this paper, we propose optimizations to the cryptography protocol Response-Based Cryptography (RBC). Since the protocol is general-purpose, it can be incorporated into post-quantum cryptography systems to authenticate users in resource-constrained environments, like Internet of Thing (IoT) devices. The optimizations proposed in this paper allow for clients to be authenticated faster. Additionally, this paper makes a cross-platform comparison of the performance of the optimized RBC protocol on the Graphics Processing Unit (GPU), the Central Processing Unit (CPU), and the Associative Processing Unit (APU). We find that the GPU and APU yield similar performance but the APU can be much more energy efficient. Furthermore, we evaluate the multi-GPU scalability of the algorithm and achieve a minimum speedup of 2.66× on 3×A100 GPUs.

ACM Reference Format:
Kaitlyn Lee, Brian Donnelly, Tomer Sery, Dan Ilan, Bertrand Cambou, and Michael Gowanlock. 2023. Evaluating Accelerators for a High-Throughput Hash-Based Security Protocol. In *52nd International Conference on Parallel Processing Workshops (ICPP-W 2023), August 7–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605731.3605745

1 INTRODUCTION

The field of cryptography has largely focused on the development of new algorithms using different mathematical techniques to improve security [2]; however, there are few cryptographic protocols that employ parallel computing. New protocols can be designed to take advantage of these resources, contrasting typical cryptographic protocols that assume that both parties have the same computational capabilities. One example is Response-Based Cryptography (RBC) [12, 13, 29, 36, 39, 40], which allows for one-time session keys. Critically, parallel computing is needed in the protocol to enable these one-time session keys and is a capability that is lacking in many canonical cryptography systems.

In typical public key infrastructure (PKI), users are assigned a public and private key pair and the private keys are stored in memory, e.g., RAM and disk. This puts the private keys at risk of being appropriated (read) by attackers. To mitigate this threat, clients can instead use Physical Unclonable Functions (PUFs) [14] to create private keys on-demand, thus avoiding storing private keys in memory. PUFs are additional hardware elements that are added to a device (or connect via USB) which act as digital fingerprints — each PUF is unique due to deviations in manufacturing. However, PUFs are well-known to produce errors relative to their initial state at production, and thus, a way to correct these errors is needed to successfully authenticate a client in PKI. Error correction codes [5, 23, 27] may be used, but low-powered Internet of Things (IoT) devices often do not have the computational power to carry out error correction, and if they were able to carry out error correction, it may leak information to an opponent. Consequently, a different method is required; instead, error correction can be performed on a secure server that has higher computational capabilities than client devices.

To enable the use of PUFs in PKI, RBC may be leveraged [12, 13, 29, 36, 39, 40], which corrects the errors generated by client PUFs. The RBC protocol is made up of two main functions: a high performance search over the PUF seed space and the generation of public keys from the PUF seeds. In previous work, the RBC search was carried out on public keys generated by AES or Post Quantum Cryptography (PQC) algorithms [29, 36, 39, 40]. There are three drawbacks of the RBC protocol outlined in prior work that we summarize as follows: (1) key generation accounts for the largest fraction of the search time and leads to high register usage which degrades performance [29], (2) the need to search over the large PUF seed space is intractable on typical multi-core CPU architectures found in servers, and (3) the method used to iterate over the PUF seed space is not well-suited for the RBC protocol.

In this paper, we propose two optimizations to the previous RBC protocol. First, instead of searching over public keys, which requires generating a public key for every PUF seed in the seed space, we

ICPP-W 2023, August 7–10, 2023, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0842-8/23/08…$15.00
https://doi.org/10.1145/3605731.3605745

propose searching over hashes instead. Since hashing is faster and requires fewer registers than most PQC public key generation functions, we show that this decreases the execution time of the RBC protocol. Second, we investigate other seed iteration methods that may be better suited for the RBC protocol than the method used in prior work. Optimizing the RBC protocol enables a higher degree of security, as clients wait less time to be authenticated and with the same time budget, the search space can be increased.

We present an evaluation of the optimized RBC protocol on mulitcore CPUs and many-core GPUs. Additionally, we include a preliminary evaluation of the Associative Processing Unit (APU) [26], with this being the first paper that evaluates the APU performance on any application. While the APU is a new accelerator architecture, it offers two potential benefits over the many-core GPU: (1) it is a compute-in-memory architecture, where data is accessed by the processor directly in-place, thus mitigating expensive off-chip memory accesses, and (2) it is often more power efficient than the GPU.

This paper is multi-faceted, as it interfaces with the fields of computer architecture, high performance computing, and both hardware and software aspects of security. Therefore, in what follows, we articulate the major objectives and contributions of the paper:

- We propose an optimized RBC protocol, RBC-Salted, that takes advantage of fast hashing and seed iteration methods to simplify the protocol, while providing the same level of security. We investigate the use of the Secure Hash Algorithm (SHA), more specifically, SHA-1 and SHA-3, in the RBC protocol. Although SHA-1 is no longer deemed secure, we include performance results for SHA-1 to provide a more thorough performance evaluation. Additionally, we do not include other hashing algorithms, such as BLAKE [4] or MD6 [37], as SHA is standardized by NIST and the other two are not.
- We investigate different PUF seed iteration methods on the GPU. These algorithms have trade-offs in terms of parallelization potential and work efficiency, where high parallelization potential is required for high throughput parallel architectures. We find that the best method results in a 22.7% reduction in authentication time. We also make an optimization to SHA-3 on the CPU and GPU.
- We present a cross-platform comparison of performance between the CPU and GPU to compare how our optimized protocol performs against the state-of-the-art. We also include a preliminary evaluation of the APU.
- We present a comparison of power consumption between the GPU and APU. This paper is the first in the literature to report APU power consumption (on any algorithm).
- We propose a multi-GPU implementation so that we can observe how the algorithm scales in multi-GPU systems, which has implications for how the algorithm may scale on other accelerators, including the APU.

The paper is organized as follows: Section 2 outlines the RBC protocol and related work, Section 3 describes the optimized protocol, Section 4 presents the evaluation, and Section 5 concludes the work.

2 BACKGROUND

2.1 Overview of Response-Based Cryptography

Motivation: In typical PKI, client devices store private keys in non-volatile memory that can be exploited by attackers. To eliminate this threat, private keys can be stored in volatile memory instead. PUFs are non-volatile memory that allow for private keys to be generated on-demand, where a new private/public key pair can be created per transaction. Thus, even if an attacker was able to recover a client's private key, it would become invalid after a short time.

Threat Model: Here we outline the security assumptions of RBC. (*i*) The server is located in a secure environment. (*ii*) The PUFs are manufactured in a secure environment. (*iii*) Once a PUF is deployed, it is in an insecure environment.

Because PUFs produce erratic bit streams, and since these bit streams are used as input to create cryptographic keys in the RBC protocol, a method to fix these erratic bits is necessary. One approach is to use helper functions and/or error correction codes to correct any bit mismatches output by the PUF on the client device. However, many low-powered IoT devices do not have the computational power to perform these correction procedures. The alternative to this approach is to use RBC, where a secure server is used to perform a parallel search over the seed space to determine whether the erratic seed produced by the client is valid [12, 15]. Consequently, novel methods for exploiting new parallel architectures is important for validating erratic PUF-generated client seeds.

The RBC Protocol: Most PKI relies on centralized certificate authorities (CAs) to authenticate and validate client public keys, which are then disseminated by a registration authority (RA). Before validation can occur, all client PUFs are enrolled in a secure facility, where the PUF image is stored in the CA. PUF images for all clients are stored in an encrypted database. During validation, PUF output is used as input into the RBC search; the bit stream from the PUF is used as input into the key generation function of a cryptographic algorithm. The public keys are then used to authenticate devices. It is important to note that client private keys are never generated or held in memory during the RBC protocol. The RBC search is described in further detail in Section 3 below.

Regarding the erratic nature of PUFs, the algorithm is agnostic to the underlying PUF hardware. However, if a PUF has an intrinsically large bit error rate, then it is conceivable that a client will not be authenticated due to an intractable search. To address this problem, the RBC protocol uses Ternary Addressable Public Key Infrastructure (TAPKI) [16]. In short, TAPKI will ignore the cells in the PUF that have a high error rate by masking them. This ensures that the RBC search is generally tractable, while still being robust to the erratic nature of PUF technology.

2.2 Complexity of the Search Process

We define the complexity of the RBC search process introduced in Section 2.1. Consider a bit stream generated from a client's PUF that has d flipped bits relative to the server's PUF image. In this paper, we assume the PUF outputs 256 bits. Thus, the upper bound

Table 1: The number of seeds searched on the server for the exhaustive (Equation 1) and average case (Equation 3) searches for each Hamming distance d up to 5.

d	1	2	3	4	5
Exhaustive	256	3.3×10^4	2.8×10^6	1.8×10^8	9.0×10^9
Average	129	1.7×10^4	1.4×10^6	9.0×10^7	4.6×10^9

number of keys u that will be searched by the server is as follows:

$$u(d) = \sum_{i=0}^{d} \binom{256}{i}. \tag{1}$$

Now, consider an opponent that would like to derive the client's private key by permuting a 256-bit bit stream and generating the key. Because the opponent does not know the starting position of the bit stream (i.e., the bits stored in the server's PUF image for the client), the worst case number of keys that need to be searched by the opponent is as follows:

$$o = \sum_{i=0}^{256} \binom{256}{i} = 2^{256}. \tag{2}$$

Comparing Equation 1 to 2, the server's search is tractable assuming d is sufficiently low (i.e., the PUF generates bit streams with limited variability), whereas the opponent's is intractable because the entire 256-bit seed space needs to be searched. Furthermore, given that the PUF allows for one-time keys, the key will change in a short amount of time.

Equation 1 illustrates the complexity of the server's search in the worst-case scenario, where all seeds up to a Hamming distance d need to be searched. However, on average, a seed will be searched halfway through the seed space at Hamming distance d, as shown in Table 1. This yields a number of seeds searched as follows:

$$a(d) = \sum_{i=0}^{d-1} \binom{256}{i} + \frac{\binom{256}{d}}{2}. \tag{3}$$

In our evaluation, we examine the performance of both exhaustive and average case searches. For the latter, we incorporate an early exit procedure that signals all threads to terminate their respective searches. This termination procedure varies depending on the hardware platform.

2.3 Related Work

Prior work can be separated into three categories: RBC, seed permutation, and hashing.

Cambou et al. [15] proposed the RBC protocol using AES and showed experimental results as a function of PUF bit error rates to assess the feasibility of performing the search under several scenarios. Cambou [12] modeled the RBC search process and showed that while a typical workstation is suitable for searches with a low bit error rate, additional computational resources are needed for PUFs that generate higher bit error rates. Philabaum et al. [36] proposed a distributed-memory CPU algorithm that was parallelized using MPI and achieved a speedup of 404× on 512 CPU cores. Wright et al. [39] parallelized the RBC search process on the GPU for the AES, ChaCha20, and SPECK block ciphers, and showed that a single Nvidia V100 GPU achieves the same search throughput as roughly 300 CPU cores, demonstrating that the GPU has superior search throughput compared to the CPU.

In addition to the above algorithms that employed AES in the search, Wright et al. [40] and Lee et al. [29] proposed using the CRYSTALS-Dilithium [19] and SABER [20] post-quantum cryptography (PQC) algorithms for the RBC search, respectively. These works presented highly efficient RBC search algorithms [29, 40]. For instance, the multi-GPU SABER [29] algorithm achieved a 2.93× speedup using 3 A100 GPUs. While these GPU-accelerated algorithms are fast, they were shown to be dramatically slower than the AES-based RBC algorithms described above, as the computational cost of AES is much lower than PQC key generation methods.

Generating the permutations for each seed in the RBC seed space is an established problem that has a number of sequential solutions available [28]. The most efficient permutation generation methods have a minimal difference from one permutation to the next while also generating the permutations in a lexicographical ordering. Early works such as Algorithm 154 [32] first solved these ordered permutation generation problems, but later works, such as Gosper's Hack [28], are able to generate the same permutations with much less work. While Gosper's Hack is highly efficient at small scale, other methods, such as Chase's Algorithm 382 [18] and Algorithm 515 [11], can scale to larger permutations without the drawbacks that Gosper's Hack encounters. While non-lexicographical permutation generation solutions are available, such as Algorithm 155 [33], they cannot be parallelized as efficiently as Chase's Algorithm 382 or Algorithm 515.

The literature on parallelizing hashing algorithms is limited on the GPU and nonexistent on the APU. We briefly describe other hashing algorithms, but focus on SHA-3, as it is standardized by NIST. SHA-3 is a subset of the Keccak [8] hashing family. Another subset of the Keccak hashing family are the tree-based algorithms. Lee et al. parallelize the tree-based Keccak hashing algorithm on the GPU [30]; however, this version of Keccak is not standardized under SHA-3. Two other works parallelize SHA-3 on the GPU, but they assign multiple threads to work on one hash at a time [17, 21]. This parallelization technique is not beneficial in the RBC protocol, as the RBC protocol requires computing many different hashes in parallel to maximize hashing throughput.

3 RBC-SALTED: THE RESPONSE-BASED CRYPTOGRAPHY PROTOCOL AND OPTIMIZATIONS

Recall from Section 2.1 that the RBC search process in the server needs to take as input the bit stream from the PUF image. Then, the search *uses a cryptographic algorithm* to generate public keys to determine whether the client should be authenticated. This scheme has one major drawback: *the logic of the search process is directly related to the cryptographic algorithm.* For instance, if a client wishes to use a different algorithm, such as a PQC algorithm (e.g., CRYSTALS-DILITHIUM [19]), the public key generation procedure in the RBC search needs to be revised. Thus, the standard RBC protocol is *algorithm aware.* Because the algorithm-aware RBC search is expensive to carry out, significant work is needed to optimize each of these cryptographic algorithms.

To address this problem, we propose RBC-Salted. Our aim is to make the search process agnostic to the type of cryptographic algorithm employed. This has several benefits, summarized as follows:

- Any cryptographic algorithm that generates public keys can be employed in the system. The algorithm could be AES or a PQC algorithm, such as one of the NIST round 4 PQC KEM and DSA candidates (Classic McEliece [9], BIKE [3], or HQC [31]) or the NIST selected algorithms (CRYSTALS-Dilithium [19], CRYSTALS-Kyber [10], FALCON [25], or SPHINCS+ [6]).
- Instead of optimizing the key generation procedure of several cryptographic algorithms to improve server key search throughput, optimization efforts can be focused on a single search method.
- A single RBC search system allows the technology to be deployed on a wider range of hardware platforms, including emerging parallel architectures. This allows us to assess the performance and other benefits of using one parallel architecture over another.

Figure 1 describes the RBC-Salted procedure, where the seed search occurs within the CA. The search begins with a client wanting to be authenticated by the CA server. The client then performs a handshake, where PUF address information is exchanged between the client and CA. The client will read the PUF at the address specified by the CA to generate a bit stream. Using the bit stream as input into SHA, the client generates a message digest, M_1. The client then sends the message digest to the CA and the CA performs the RBC search. We describe the RBC search as follows:

(1) The server generates a 256-bit seed, S_{init}, using the client's PUF image. The seed is used as input to the RBC search.
(2) S_{init} is hashed using SHA (this can be any variant of SHA) to create a message digest, M.
(3) Using the message digest M_1 sent to the server from the client, the server checks if $M = M_1$.
(4) If $M \neq M_1$, the RBC search is conducted, starting at $d = 1$, where d bits are flipped in the bit stream, and the CA hashes the new seed S to generate the next message digest.
(5) If the CA finds message digests (M and M_1) that match at a Hamming distance d, then the client is authenticated.
(6) If the CA does not find message digests (M and M_1) that match, the Hamming distance d is increased by 1, and the algorithm starts again at step 4 above.
(7) Once the client is authenticated, i.e., the server found the client's 256-bit seed, which occurs when $M = M_1$, the seed S is salted (e.g., S is bit shifted) to create S'.
(8) The public key for the client P_{k1} is generated using S' with a cryptographic algorithm.
(9) The RA is updated using the client's public key.

A critical element of this procedure is that both the client and server share the same salt, such that there is not a correspondence between the public key and the message digests.

Additionally, RBC uses a time threshold, T, for which it must authenticate a client. Because the error rate could potentially be high if the client generates a bit stream from a PUF with a high error rate, the search will be intractable, as the complexity scales exponentially with d. If a timeout occurs, the CA simply sends the client a new PUF address and the process is restarted. In this paper, we set $T = 20$ s, as proposed by prior work [29].

Observe that in RBC-Salted, the server only needs to generate a public key once. In contrast, the original RBC algorithm needs to generate public keys for each permutation of the 256-bit seed. Because key generation in most cryptographic algorithms is more expensive than hashing, the salted approach is less expensive than the original RBC protocol.

Algorithm 1 RBC-Salted Search Algorithm

```
1:  procedure SALTED-CPU(d, M_1, S_init)
2:      r ← getThreadId()
3:      p ← getNumThreads()
4:      if r = 0 then
5:          M ←SHA(S_init)
6:          if M = M_1 then
7:              NotifyAllThreadsToExitSearch()
8:              return True
9:      for i ∈ 1, ..., d do
10:         n ← C(256, i)/p
11:         for j ∈ 1, ..., n do
12:             S ← genNextSeed(S_init, r, n)
13:             M ← SHA(S)
14:             if M = M_1 then
15:                 NotifyAllThreadsToExitSearch()
16:                 return True
17:     return False
```

3.1 Overview of the Algorithm

We present an overview of RBC-Salted using pseudocode, providing minimal hardware-specific details. In the following subsections, we describe how the work is assigned to the processing elements (PEs) across several hardware platforms. We use PE to denote a core on the CPU and GPU and a set of cores on the APU.

Recall that the RBC search response time is bounded by the PUF error rate and the authentication threshold T. Using benchmarks, we compute the largest value of d that yields a latency $\leq T$.

For the purposes of presentation, we assign each PE $n = N/p$ seeds to search at each Hamming distance, d, where $N = \binom{256}{d}$ and p is the number of PEs.

The pseudocode of the search process is given in Algorithm 1. **Algorithm input:** a maximum Hamming distance, d, for which to search, the message digest received from the client M_1, and the seed S_{init} that is retrieved from the server's client PUF image. **Algorithm output:** returns true if the client is authenticated, and false otherwise. The algorithm begins by retrieving a thread identifier, r, and the total number of threads p (lines 2–3). On lines 4–7, the first thread ($r = 0$) checks a Hamming distance of 0 by using SHA to hash the seed S_{init} to generate the message digest, M. If the message digests match ($M = M_1$), then the client is authenticated and all threads exit. Otherwise, a parallel search needs to proceed.

The search is executed in a data-parallel fashion, where each thread searches a different domain of the total seed space. Line 9 loops over each Hamming distance up to d. The number of seeds each thread needs to search for Hamming distance i is obtained on line 10, and a loop is entered on line 11. Inside the loop, a seed is obtained by calling the function *genNextSeed()*, which requires the initial seed, S_{init}, the thread id, r, and the number of seeds to search, n. With this information, the function computes the next seed, S, that the thread will search, where each thread searches a different disjoint subspace of the total seed space. The seed is then hashed on line 13 to create the message digest M. If the server and client's message digests match ($M = M_1$), all threads exit and the client is authenticated. If the seed is not found when searching up to d, then the client is not authenticated. In practice, the client and server would start the process again with a new handshake.

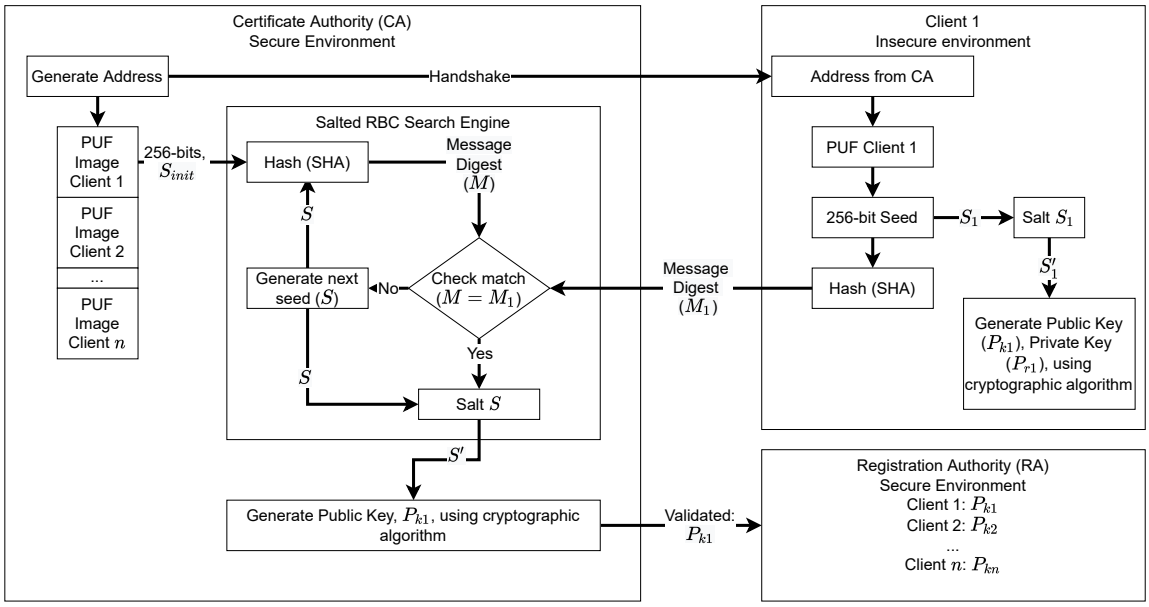

Figure 1: The RBC-Salted protocol. The cryptographic algorithm can be replaced with any public key cryptographic protocol. See text for details.

The exit procedure described in Algorithm 1 on lines 7 and 15 is dependent on the hardware platform. We describe the exit procedures and algorithm-specific optimizations in greater detail in the following subsections.

3.2 Salted-GPU

Here we present the GPU algorithm. In Algorithm 1, on lines 2–3, the GPU algorithm computes r and p using built-in variables in CUDA, where `r=threadIdx.x+blockIdx.x*gridDim.x`, and `p=gridDim.x*blockDim.x`. The loop on line 9 is executed on the host, where a kernel is launched to process a single Hamming distance (i.e., if we search $d = 5$, then we launch 5 kernels).

Section 3.1 describes each PE being assigned $n = \binom{256}{d}/p$ seeds to search at each Hamming distance d. Since the GPU typically requires thread oversubscription to hide memory latency and stalls, the value of p will be several times larger than the number of GPU cores. In our evaluation, we tune the parameter p for the highest Hamming distance searched.

Early Exit: The exit procedure described in Algorithm 1 on lines 7 and 15 uses a flag in unified memory such that it is accessible on the host and across multiple GPUs. The host requires the flag such that it knows whether to terminate the search early or to launch the kernel for the next Hamming distance. The GPU threads require the flag to know whether to return from the kernel early. The thread that finds the correct seed atomically updates the flag. After each thread processes a seed, the flag is checked to see whether it has been set. If it has, the thread returns.

We present three optimizations made to our seed iteration, hashing, and memory storage, in the follow subsections.

3.2.1 Seed Permuting Optimization. Seed permutations are generated by flipping d-bits of S_{init}, as determined by a combination. Two different combination generation methods are evaluated with a focus on the efficiency of parallel generation of combinations. While there is a large body of literature addressing combination generation [28, 34, 38], there is little that addresses parallel combination generation and performance of the methods vary and can significantly impact overall response time. For example, Gosper's Hack [28], as used in prior work [29, 36, 39, 40], is a popular combination generator that performs well with native datatypes, e.g. 8-bit, 32-bit, and 64-bit types. However, 256-bit input seeds cannot be stored in a native datatype. Consequently, the performance is poor. Instead, we evaluate two algorithms that are both highly optimized for sequential execution. We also compare them to Gosper's hack, described above.

Algorithm 515: One way of generating a unique combination is to pull a single combination from all possible combinations based on an index in that ordering. This allows a combination to be generated without repetition based on a single iteration value. This method parallelizes well because combinations can be generated independently of each other. Also, the method allows for pre-computing a common subset of the computation and then using a lookup table to exploit the high memory bandwidth available on the GPU. Algorithm 515, developed by Buckles and Lybanon [11] and adapted for the GPU, is evaluated in Section 4.5.

Chase's Algorithm 382: Gray codes generate combinations by minimizing the work needed to transition from one combination to the next, while avoiding repetition and maintaining an order [28, 38]. Gray codes are often computed recursively [24]; however, this is impossible for $\binom{256}{5}$ combinations when executed on GPUs due to recursive depth limitations. Chase's Algorithm 382 [18] is a non-recursive Gray code that sequentially generates combinations. Each new combination is dependent on the state of the previous combination, making it difficult to parallelize. To circumvent this problem, we modify Algorithm 382 such that the combinations are generated sequentially, with a number of combination states being

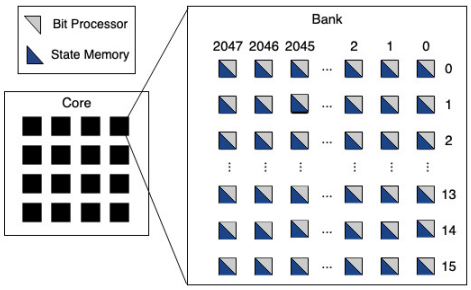

Figure 2: Representation of a GSI Gemini APU core consisting of 16 banks. Each bank is made up of 2048×16 bit processors (top-right, gray) that are coupled with state memory (bottom-left, blue).

saved at regular intervals. The number of saved states corresponds to the number of desired independent threads used for hashing. Each state is evenly spread throughout the combination sequence so that threads have equal workloads. To compute the combinations on the GPU, the array of saved states is sent to the GPU and threads are assigned an initial state. Each thread then uses Algorithm 382 to generate and hash the seed permutation while maintaining a single state per thread. While overhead may make Algorithm 382 seem like a less attractive option compared to Algorithm 515, the reduction in computation of Algorithm 382 leads to a speedup over Algorithm 515. Since the set of initial states can be loaded into GPU memory once and used to authenticate all clients, this overhead is excluded in the evaluation.

3.2.2 Hashing Optimizations. The SHA-3 algorithm from Bertoni et al. [7] was optimized for use in the RBC scheme by eliminating unnecessary functionality in the generic implementation. Most hashing is designed for variable sized inputs, which we do not require for hashing 256-bit seeds. As such, we fixed the padding bits for our 256-bit seeds to reduce several conditional statements. We found that this improved the performance of Salted-GPU by ~3%. We use this optimization in all GPU experiments in Section 4.

3.2.3 Storing Chase's Algorithm 382 State in Shared Memory. Recall from Section 3.2.1 that Chase's algorithm stores a state for each thread. In order to reduce memory latency, we store the state in shared memory [22]. This results in 1.20× and 1.01× speedups for SHA-1 and SHA-3, respectively. We use this optimization in all GPU experiments in Section 4.

3.3 Salted-APU

We present the APU, Salted-APU, and its optimizations. The APU used in our experiments contains 4 cores. As shown in Figure 2, each core is made up of 16 banks, and each bank consists of 2048×16 bit processors (BPs). Thus, each chip consists of $4 \times 16 \times 2048 \times 16 \approx 2$ million BPs. For the APU, a PE is software defined based on the number of BPs needed for each thread. Similarly to the GPU, an efficient APU program must account for the memory needed by the algorithm, in addition to maximizing resource (core) utilization. The number of BPs that are active is dependent on the amount of state memory used (akin to registers on a CPU).

Each PE is defined by combining 32 and 80 BPs for SHA-1 and SHA-3, respectively, as SHA-3 has a greater state footprint than SHA-1. This means that 2.5× more PEs can run concurrently for SHA-1 than for SHA-3. In total, there are $4 \times 16 \times \frac{2048}{2} = 65$k PEs for SHA-1 and $4 \times 16 \times \lfloor \frac{2048}{5} \rfloor = 26$k PEs for SHA-3.

In Algorithm 1, on line 3, we define the number of threads p as the number of APU PEs. The loop on line 11 is executed on each PE and starts by loading startup combinations for the seed iterator. Each combination is used to generate the next seed permutation S from S_{init}. In total, each startup combination is used to generate 256 seed permutations, after which a new startup seed is loaded for the next batch of computation.

Early Exit: The exit procedure described in Algorithm 1, on lines 7 and 15 uses a flag stored in associative memory that all threads have access to. Once the client's seed is found, the flag is updated. All threads check the flag after completion of the current 256 seed permutation batches.

3.3.1 Seed Permutation. In contrast to the GPU, the design space for seed permutation algorithms is smaller for the APU, as the allocation of PEs is highly coupled to the algorithm. Consequently, we evaluate a novel seed permutation method that is designed specifically for the APU architecture. Other methods used with the GPU and CPU, such as Chase's Algorithm 382 and Algorithm 515, would be bottlenecked by PE allocation and excessive memory usage on the APU.

3.4 Salted-CPU

Salted-CPU is parallelized with OpenMP. The seed permutation uses the same methodology as described in Section 3.2.1, where Algorithm 1 shows how RBC-Salted is designed for the CPU.

Early Exit: The early exit procedure shown in Algorithm 1 on lines 7 and 15 uses a flag that is stored in main memory to signal all of the threads to stop searching.

Optimizations: The CPU implementation of RBC-Salted uses the same optimizations to SHA-3 that were used for the GPU, as well as the same seed generation optimizations.

4 EXPERIMENTAL EVALUATION

Table 2 summarizes relevant notation defined in prior sections.

Table 2: Summary of notation that appears in the evaluation.

	Definition
d	The searched Hamming distance.
T	The time threshold for authentication. We set $T = 20$ s.
p	The number of threads executed for a given algorithm.
n	The number of seeds searched per thread at a given d.
b	Number of CUDA threads per block.

4.1 Experimental Methodology

Salted-GPU is written in C/C++ and CUDA, Salted-CPU is written in C, and Salted-APU is written using C++ and APL (APU assembly language). We summarize the platforms that we use in our experiments in Table 3.

As described in Section 3, the complexity of the RBC search is bound by the Hamming distance d. In our experiments, where we show the RBC search-only response times (excluding communication), we present both upper bound (exhaustive search) and average case performance. Because the PUF generates errors, the seeds are

Table 3: Platform Details: the cores and memory columns for the GPU and APU refer to the number of cores per device.

	CPU				GPU or APU				
Platform	Model	Total Cores	Clock	Memory	Model	Cores	Clock	Memory	Software
PlatformA	2×AMD EPYC 7542	2×32 (64)	2.9 GHz	512 GiB	1×NVIDIA A100	6912	1410 MHz	40 GiB	CUDA 11
PlatformB	Intel i7-7700	4	3.6 GHz	32 GiB	Gemini APU	131072	575 MHz	4 GiB	APL

stochastic; therefore, when we show the average case performance, we present an average of 1,200 trials.

A typical bit error rate from the PUF is 5 bits, and if it is lower, we perform noise injection on the client to ensure that we have flipped 5 bits in the seed; consequently, we search Hamming distances up to $d = 5$. Throughout the evaluation, we will refer to searching a Hamming distance, and we highlight that, as shown in Equations 1 and 3 and Table 1, we search all Hamming distances up to and including the value of d. Thus, when we refer to searching $d = 5$, this implies that we search the following values: $d \in \{0, 1, \ldots, 5\}$.

4.2 Summary of Experimental Scenarios

We present a comparison of RBC-Salted using two hashing algorithms: SHA-1 and SHA-3. Although SHA-1 is no longer deemed secure, we include performance results for SHA-1 to provide a more thorough performance evaluation when comparing the algorithms tailored for the CPU, GPU, and APU architectures. Additionally, we only investigate the use of SHA and do not include other hashing algorithms, such as BLAKE [4] or MD6 [37], as SHA is standardized by NIST and the other two are not. As described in Section 3, we use a $T = 20$ s time threshold in this paper, consistent with prior work [29].

We perform an end-to-end search, requiring network communication between a client equipped with a PUF and a server that performs the RBC search. For Salted-GPU and Salted-CPU results, both client and server are located in the U.S.A. For Salted-APU, the server is located in Israel. Thus, to make a fair comparison between algorithms, we report the network latency derived from the CPU and GPU experiments, and not the latency between the U.S.A. and Israel. We also present results without any of the end-to-end communication costs, where we only report the response time required to perform the RBC search.

4.3 Implementation Configurations

We compare the performance of Salted-GPU, Salted-CPU, and Salted-APU for $d = 5$. We outline the configurations of the implementations for most of the experiments, as follows. For each algorithm, we select the parameters that achieve the best performance.

Salted-GPU: This algorithm is executed on PlatformA using 1×A100 and is configured with $n = 100$ seeds searched per thread and $b = 128$ threads per block.

Salted-APU: This algorithm is executed on PlatformB. As mentioned in Section 3.3, SHA-1 and SHA-3 use 65k and 26k PEs, respectively.

Salted-CPU: This algorithm is executed on PlatformA. We execute Salted-CPU using $p = 64$ CPU threads, matching the number of physical cores on our platform. We achieve speedups of 59× and 63× on 64×CPU cores using SHA-1 and SHA-3, respectively.

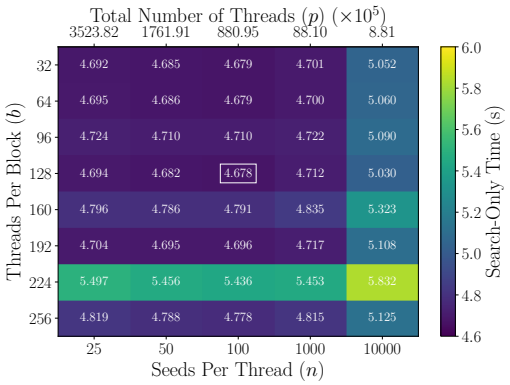

Figure 3: Heatmap showing the search-only time in seconds for different combinations of seeds per thread (n) and threads per block (b) for an exhaustive search with SHA-3 and $d = 5$. We also include the total number of threads required to run the RBC search with the given seeds per thread value. The minimum search time is reached with 100 seeds per thread and 128 threads per block, outlined in white.

We also show the timing threshold, $T = 20$ s (dashed, horizontal line). We find that Salted-CPU does not obtain authentication within this time limit using SHA-3.

4.4 Salted-GPU: Selection of the Number of Seeds Searched Per Thread and Seeds Iterated Between Match Checks

To achieve the best performance, the GPU algorithm needs to use a sufficient number of threads to saturate its resources, while not creating too many threads that could cause non-negligible overhead (an exhaustive search at our nominal Hamming distance, $d = 5$, requires generating over 8 billion seeds, so assigning a single thread per seed will cause overhead). Therefore, employing Salted-GPU using SHA-3, we perform a grid search for the best number of seeds assigned per thread as a function of the number of threads per block. We find that the best performance is obtained when we select $n = 100$ seeds per thread and $b = 128$ threads per block on PlatformA. Furthermore, several sets of parameters achieve similarly good performance, so the number of threads per block and seeds per thread can be selected in a large range. We omit showing the results for SHA-1, as they are similar.

Recall in Algorithm 1 on lines 7 and 15, a flag stored in unified memory is used to notify threads that the client's seed has been found, i.e., the flag is set to true. Accessing this flag after every seed iteration could have negative effects on performance. For this reason, we increased the number of seeds iterated between checks from 1 up to 64 and found that increasing the iterations did not have any performance impact. Thus, we check if the client's hash has been found after every seed iteration in the rest of this paper.

Table 4: The total exhaustive search-only time (s) for three seed iterator methods. The search was conducted for $d = 5$ using SHA-3 on PlatformA using one GPU. Since we change the algorithm, we use the best parameters (p, n, and b) for each method, similarly to Figure 3.

Algorithm	Search-Only Time
Alg. 382 (Sec. 3.2.1)	4.67
Alg. 515 (Sec. 3.2.1)	7.53
Prior Work [29, 39, 40]	6.04

4.5 Salted-GPU: Seed Iterators

Prior work in RBC [29, 39, 40] used a modified Gosper's Hack for generating seed permutations. Recall from Section 3.2.1 that Gosper's Hack is efficient, but only if the size of the seed (256-bit in RBC-Salted) is a native integer datatype (only up to 64-bit on current GPUs). Adjusting Gosper's Hack to work with a non-native datatype significantly reduces performance.

Instead, there are multiple methods for generating permutations of 256-bit seeds. Recall from Section 3.2.1, that Algorithm 382 is inherently sequential but minimizes work, whereas, Algorithm 515 is highly parallelizable, but requires more work to permute the seeds. Another approach in the latter class that has high parallelization potential, is that proposed in prior work on RBC searches [29, 39, 40], Gosper's Hack. We find that despite Algorithm 515 and Gosper's Hack having excellent parallelization potential, they perform much worse than using Algorithm 382 (Chase's sequence) when optimized for GPU execution. Algorithm 382 obtains speedups of 5.89× and 6.77× over Algorithm 515 and Gosper's Hack, respectively, when using SHA-3. In summary, we find that it is best to optimize the work-efficient sequential algorithm, rather than using the less efficient but highly parallelizable algorithm. Note that we do not time the seed iteration separately from SHA-3, as they execute in the same GPU kernel and so there is a dependency between these two components.

4.6 Comparison of Algorithm End-to-End and Search-Only Performance

We compare the total end-to-end response time of the algorithms, showing the breakdown of the time needed for network communication, including the client reading the 256-bit stream from the PUF (the client is a laptop and the PUF is connected via a USB port), and the search time. We combine PUF reading time and network communication time.

Table 5 shows the performance of the three algorithms for both SHA-1 and SHA-3, across the exhaustive and average-case search scenarios. Here, we show searching up to $d = 5$. Comparing Salted-GPU to Salted-APU, using SHA-1, we find that the algorithms yield roughly equivalent search throughput, where Salted-GPU achieves a speedup of 1.02× for the exhaustive search over Salted-APU, but a slowdown of 0.99× for the average-case search. In contrast, Salted-CPU with $p = 64$ is significantly slower, where the GPU obtains a speedup of 5.54× and 3.97× on exhaustive and average case searches, respectively, over Salted-CPU.

In contrast to SHA-1, we find that SHA-3 is much more efficient on the GPU than the APU, where Salted-GPU obtains a speedup over Salted-APU of 2.99× and 2.91× on the exhaustive and average case searches, respectively. Similarly, the speedup of Salted-GPU over Salted-CPU is 13.06× (exhaustive) and 12.61× (average case).

Table 5: End-to-end response time (s) of Salted-GPU, Salted-APU, and Salted-CPU for $d = 5$ and SHA-1 and SHA-3. Exhaustive and average case searches refer to searching the seeds defined by Equations 1 and 3, respectively.

Algorithm	Search Type	Comm. Time	Search Time	Total Time
SHA-1				
Salted-GPU	Exhaustive	0.90	1.56	2.46
Salted-APU	Exhaustive	0.90	1.62	2.52
Salted-CPU	Exhaustive	0.90	12.09	12.99
Salted-GPU	Average	0.90	0.85	1.75
Salted-APU	Average	0.90	0.83	1.73
Salted-CPU	Average	0.90	6.04	6.94
SHA-3				
Salted-GPU	Exhaustive	0.90	4.67	5.57
Salted-APU	Exhaustive	0.90	13.95	14.85
Salted-CPU	Exhaustive	0.90	60.68	61.58
Salted-GPU	Average	0.90	2.42	3.32
Salted-APU	Average	0.90	7.05	7.95
Salted-CPU	Average	0.90	30.52	31.42

Table 6: Salted-GPU and Salted-APU search-only energy consumption of the exhaustive search for a Hamming distance $d = 5$.

Algorithm	SHA Version	Total Joules	Maximum Watts	Idle Watts
Salted-GPU	1	317.20	253.43	31.53
Salted-APU	1	124.43	83.81	22.10
Salted-GPU	3	946.55	258.29	31.53
Salted-APU	3	974.06	83.63	22.10

Thus, Salted-GPU is highly proficient at hashing the more expensive SHA-3 algorithm and gains a runtime performance advantage over the CPU and APU. Recall from Section 4.3, that for Salted-APU, SHA-3 requires 5 BPs per PE, whereas SHA-1 requires 2 BPs; therefore, SHA-3 is more resource intensive. This explains the performance degradation when executing SHA-1 vs. SHA-3 on the APU.

In summary, we find that we can exhaustively search $d = 5$ and obtain authentication within the nominal authentication time threshold [29] of $T = 20$ s for all algorithms on SHA-1, whereas we are above the threshold time only for the Salted-CPU algorithm using 64 cores on SHA-3.

4.7 Comparison of GPU and APU Energy Footprints

One motivation for using the GPU is that it has lower energy consumption than the CPU for many applications. Similarly, we compare the energy and power footprints of the Salted-GPU and Salted-APU algorithms. Unlike the GPU, the APU is designed for in-memory processing. It is known that most of the energy in many programs is consumed by moving data between memory and processor [1, 35], and in the APU, this is nearly eliminated. Table 6 shows the total energy (J) needed to perform an exhaustive RBC search for $d = 5$. The table also shows the maximum and idle wattages for each architecture, where idle watts refers to the device being powered without program execution. In all presented energy measurements, we include this idle energy. We find that for SHA-1, the APU is superior to the GPU in terms of energy consumption, requiring only 39.2% of the joules needed by the GPU. Regarding SHA-3, the energy consumption is roughly equivalent, as the GPU

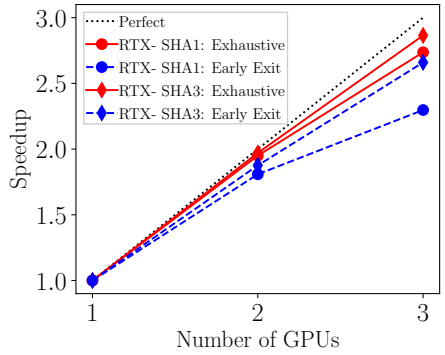

Figure 4: Multi-GPU scalability of the search-only time, showing the speedup of up to 3xA100 GPUs in PlatformA for SHA-1 (circle markers) and SHA-3 (diamond markers) for exhaustive (solid red lines), and early exit (dashed blue lines) searches. We use the best parameters (p, n, and b) for each number of GPUs, similarly to Figure 3, as these parameters change when increasing the number of GPUs.

Table 7: Comparison of execution time (s) for previous RBC work [29, 39, 40] and this work on 64xCPU Cores, 3xA100 GPUs, and the APU.

		CPU/GPU (PlatformA)			APU (PlatformB)	
Ref.	Algorithm	d	CPU Time (s)	GPU Time (s)	d	Time (s)
[39]	AES-128	5	44.7	2.56	-	-
[29]	LightSABER	4	44.58	14.03	-	-
[40]	Dilithium3	4	204.92	27.91	-	-
This Work	SHA-3	5	60.68	4.67	5	13.95

is faster than the APU, which makes the APU have a similar energy footprint to the GPU.

4.8 Salted-GPU: Multi-GPU Scalability

All performance results reported above conducted the RBC search using a single A100 GPU. Here, we assess multi-GPU scalability on our GPU platform with 3xA100 GPUs.

Figure 4 plots the speedup of Salted-GPU on up to 3xA100 GPUs for SHA-1 and SHA-3, for both exhaustive and early exit searches. We find that the exhaustive SHA-1 and SHA-3 searches have the best scalability; SHA-3 obtains a speedup of 2.87× on 3xA100 GPUs, and therefore, exhibits excellent scalability. In contrast, the early exit searches have lower parallel efficiency, where we observe that SHA-3 obtains a 2.66× speedup on 3xA100 GPUs, indicating that the early exit procedure has non-negligible overhead. We also find that for a given search type (exhaustive or early exit), the scalability of SHA-3 is greater than SHA-1. This is expected, as SHA-3 is more compute-intensive than SHA-1, and so it can better exploit resources with an increasing number of GPUs. Because of the similarities between the GPU and APU, the high scalability of the GPU has implications that the algorithm will scale well on the APU.

4.9 Comparison with the state-of-the-art

Here, we compare this work to previous RBC work [29, 39, 40]. Table 7 summarizes the time to authenticate a client using the various RBC methods. For comparison purposes, we executed the algorithms from prior work on PlatformA. For each RBC method, we give the cryptographic algorithm used, the platform used, the Hamming distance d searched, the CPU time (s), the GPU time (s), and the parameters used to gather results on the APU from this work. Because RBC-Salted only requires generating a PQC key once (after the seed has been found), and instead relies on hashing and salting to perform the search, we see that Salted-GPU outperforms both PQC RBC works on the GPU [29, 40], as it takes both works over 5 seconds to search up to $d = 4$, while Salted-GPU can search up to $d = 5$ in under 5 seconds. Additionally, Salted-APU outperforms both PQC RBC works on the GPU [29, 40]. Comparing Salted-GPU to the AES RBC method, we find that the AES method is $\approx 45.2\%$ faster; however, because AES is symmetric, SHA-3 is a one-way function, and an asymmetric key generation function can be used at the end of the RBC search, RBC-Salted supplies more security. Overall, because RBC-Salted is more efficient than the PQC RBC methods [29, 40], it enables searching larger Hamming distances, allowing the algorithm able to authenticate clients with a higher PUF bit error rate, in addition to making the protocol more secure.

5 DISCUSSION & CONCLUSIONS

Very few cryptographic protocols take advantage of parallel computing, which is a missed opportunity for designers of cryptographic protocols. RBC places the burden of authentication on a server located in a secure environment, and thus obviates the need to have low-powered client devices perform error correction. In this paper, we made two optimizations to the RBC protocol: using a fast hashing algorithm and salting technique so that the algorithm is no longer algorithm-aware, and using a more suitable seed iteration method tailored to the target platforms. With these optimizations, we proposed RBC-Salted and evaluated the performance of RBC-Salted on CPUs, GPUs, and APUs.

We made a cross-platform evaluation of the optimized RBC protocol on three architectures: the CPU, GPU, and APU. We investigated the use of an APU that allows for accessing data in-place. We found that Salted-GPU and Salted-APU perform nearly identically when hashing using SHA-1, but the GPU outperforms the APU when hashing with SHA-3. Both the GPU and APU outperform the CPU for both SHA versions. We furthered our comparison of the GPU and the APU by investigating energy consumption. We found that the APU has superior energy efficiency for SHA-1, but the total joules for SHA-3 is nearly equivalent between the APU and GPU because of the longer search time required of the former architecture. Although the GPU outperformed the APU for SHA-3, the APU shows promising results and has potential use in other applications.

Future work includes examining multi-APU scalability within a single node, where 8×APU can be installed within the 2U form factor since the APU has a smaller form factor than the A100 GPU. This may enable the APU to have better single node scalability than the GPU. Additionally, since we observed that Salted-CPU achieved near-perfect parallel efficiency on 64 CPU cores, another direction is to scale the multi-core CPU algorithm across multiple compute nodes in a cluster. Finally, since Salted-GPU is able to authenticate a client well under the $T = 20$ s timing threshold, we can purposefully inject noise into the client's PUF output, thereby

increasing the Hamming distance that needs to be searched by the server, further increasing the level of security afforded by RBC.

ACKNOWLEDGMENTS

Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-23-2-0014. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute Caches. In *2017 IEEE Intl. Symp. on High Performance Computer Architecture*. 481–492.

[2] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, et al. 2022. Status report on the third round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST* (2022).

[3] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, France Worldline, Loïc Bidoux, Olivier Blazy, Philippe Gaborit, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, et al. 2021. Bike. *Round 3 submission to the NIST PQC Project* (2021).

[4] Jean-Philippe Aumasson, Willi Meier, Raphael C-W Phan, and Luca Henzen. 2014. The hash function BLAKE. In *Information Security and Cryptography*. Springer.

[5] Georg T Becker, Alexander Wild, and Tim Güneysu. 2015. Security Analysis of Index-Based Syndrome Coding for PUF-Based Key Generation. In *2015 IEEE Intl. Symp. on Hardware Oriented Security and Trust*. 20–25.

[6] Daniel J Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. 2019. The SPHINCS+ signature framework. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2129–2146.

[7] Guido Bertoni, Joan Daemen, Michaël Peeters, GV Assche, and RV Keer. 2012. 1001 Ways to Implement Keccak. In *Third SHA-3 candidate Conf.*

[8] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2013. Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 313–314.

[9] Bhaskar Biswas and Nicolas Sendrier. 2008. McEliece Cryptosystem Implementation: Theory and Practice. In *Post-Quantum Cryptography*, Johannes Buchmann and Jintai Ding (Eds.). Springer Berlin Heidelberg, 47–62.

[10] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS-Kyber: a CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symp. on Security and Privacy*. 353–367.

[11] Bill P. Buckles and Matthew Lybanon. 1977. Algorithm 515: Generation of a vector from the lexicographical index [G6]. *ACM Transactions on Mathematical Software* 3, 2 (1977), 180–182.

[12] B. Cambou. 2019. Unequally Powered Cryptography With Physical Unclonable Functions for Networks of Internet of Things Terminals. In *2019 Spring Simulation Conf.* 1–13.

[13] Bertrand Cambou, Michael Gowanlock, Bahattin Yildiz, Dina Ghanaimiandoab, Kaitlyn Lee, Stefan Nelson, Christopher Philabaum, Alyssa Stenberg, and Jordan Wright. 2021. Post Quantum Cryptographic Keys Generated with Physical Unclonable Functions. *Applied Sciences* 11, 6 (2021).

[14] Bertrand Cambou and Saloni Jain. 2022. Key Recovery for Content Protection Using Ternary PUFs Designed with Pre-Formed ReRAM. *Applied Sciences* 12, 4 (2022), 1785.

[15] Bertrand Cambou, Christopher Philabaum, Duane Booher, and Donald A Telesca. 2019. Response-Based Cryptographic Methods with Ternary Physical Unclonable Functions. In *Future of Information and Communication Conf.* Springer, 781–800.

[16] B Cambou and D Telesca. 2018. Ternary Computing to Strengthen Information Assurance. Development of Ternary State Based Public Key Exchange. In *IEEE, SAI-2018, Computing Conf.*

[17] Pierre-Louis Cayrel, Gerhard Hoffmann, and Michael Schneider. 2011. GPU Implementation of the Keccak Hash Function Family. In *Information Security and Assurance*, Vol. 200. 33–42. https://doi.org/10.1007/978-3-642-23141-4_4

[18] Phillip J Chase. 1970. Algorithm 382: Combinations of m out of n objects [g6]. *Communications of the ACM* 13, 6 (1970), 368.

[19] CRYSTALS-Dilithium. 2022. *CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation. Part of the Round 3 Submission Package to NIST.*

[20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. 2018. Saber: Module-LWR Based Key Exchange, CPA-Secure Encryption and CCA-Secure KEM. In *Progress in Cryptology – AFRICACRYPT 2018*, Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi (Eds.). Springer Intl. Publishing, Cham, 282–305.

[21] Thuong Nguyen Dat, Keisuke Iwai, and Takakazu Kurokawa. 2017. Implementation of High Speed Hash Function Keccak Using CUDA on GTX 1080. In *2017 Fifth International Symposium on Computing and Networking (CANDAR)*. 475–481. https://doi.org/10.1109/CANDAR.2017.47

[22] Danilo De Donno, Alessandra Esposito, Luciano Tarricone, and Luca Catarinucci. 2010. Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD [EM Programmer's Notebook]. *IEEE Antennas and Propagation Magazine* 52, 3 (2010), 116–122.

[23] Jeroen Delvaux, Dawu Gu, Dries Schellekens, and Ingrid Verbauwhede. 2014. Helper Data Algorithms for PUF-Based Key Generation: Overview and Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 6 (2014), 889–902.

[24] Peter Eades and Brendan D. McKay. 1984. An Algorithm for Generating Subsets of Fixed Size with a Strong Minimal Change Property. *Inform. Process. Lett.* 19, 3 (1984), 131–133.

[25] Falcon. 2022. *Falcon: Fast-Fourier Lattice-Based Compact Signatures over NTRU, Specification v1.2.*

[26] GSI Technologies. 2020. *An Associative Processing Structure Challenges von Neumann Architecture for Latency and Energy Efficiency.* Retrieved April 6, 2022 from https://www.gsitechnology.com/sites/default/files/Presentations/GSIT-Gemini-APU-Tech-Paper.pdf

[27] Maximilian Hofer and Christoph Böhm. 2010. Error Correction Coding for Physical Unclonable Functions. In *Austrochip, Workshop on Microelectronics*.

[28] Donald Ervin Knuth. 2011. *The Art of Computer Programming*. Vol. 4. Pearson Education.

[29] Kaitlyn Lee, Michael Gowanlock, and Bertrand Cambou. 2021. SABER-GPU: A Response-Based Cryptography Algorithm for SABER on the GPU. In *2021 IEEE 26th Pacific Rim Intl. Symp. on Dependable Computing*. 123–132.

[30] Wai-Kong Lee, RaphaëL C.-W Phan, Bok-Min Goi, Lanxiang Chen, Xiujun Zhang, and Naixue N. Xiong. 2018. Parallel and High Speed Hashing in GPU for Telemedicine Applications. *IEEE Access* 6 (2018), 37991–38002. https://doi.org/10.1109/ACCESS.2018.2849439

[31] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loıc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and IC Bourges. 2018. Hamming quasi-cyclic (HQC). *NIST PQC Round* 2, 4 (2018), 13.

[32] Charles J Mifsud. 1963. Algorithm 154: Combination in lexicographical order. *Commun. ACM* 6, 3 (1963), 103.

[33] Charles J Mifsud. 1963. Algorithm 155: Combination in any order. *Commun. ACM* 6, 3 (1963), 103.

[34] William H Payne and Frederick M Ives. 1979. Combination Generators. *ACM Transactions on Mathematical Software* 5, 2 (1979), 163–172.

[35] Ardavan Pedram, Stephen Richardson, Mark Horowitz, Sameh Galal, and Shahar Kvatinsky. 2017. Dark Memory and Accelerator-Rich System Optimization in the Dark Silicon Era. *IEEE Design & Test* 34, 2 (2017), 39–50.

[36] Christopher Philabaum, Christopher Coffey, Bertrand Cambou, and Michael Gowanlock. 2021. A Response-Based Cryptography Engine in Distributed-Memory. In *Intelligent Computing*, Kohei Arai (Ed.). Springer Intl. Publishing, Cham, 904–922.

[37] Ronald L Rivest, Benjamin Agre, Daniel V Bailey, Christopher Crutchfield, Yevgeniy Dodis, Kermin Elliott Fleming, Asif Khan, Jayant Krishnamurthy, Yuncheng Lin, Leo Reyzin, et al. 2008. The MD6 hash function–a proposal to NIST for SHA-3. *Submission to NIST* 2, 3 (2008), 1–234.

[38] Carla Savage. 1997. A Survey of Combinatorial Gray Codes. *SIAM review* 39, 4 (1997), 605–629.

[39] Jordan Wright, Zane Fink, Michael Gowanlock, Christopher Philabaum, Brian Donnelly, and Bertrand Cambou. 2021. A Symmetric Cipher Response-Based Cryptography Engine Accelerated Using GPGPU. In *2021 IEEE Conf. on Communications and Network Security*. 146–154.

[40] Jordan Wright, Michael Gowanlock, Chistopher Philabaum, and Bertrand Cambou. 2022. A CRYSTALS-Dilithium Response-Based Cryptography Engine Using GPGPU. In *Proceedings of the Future Technologies Conf. 2021, Volume 3*, Kohei Arai (Ed.). Springer Intl. Publishing, Cham, 32–45.