A CRYSTALS-Dilithium Response-Based Cryptography Engine using GPGPU*

Jordan Wright[1], Michael Gowanlock[1], Chistopher Philabaum[1], and Bertrand Cambou[1]

School of Informatics, Computing, & Cyber Systems, Northern Arizona University, Flagstaff AZ 86011, USA

Abstract. Post-quantum cryptography (PQC) will be needed to secure public-key cryptosystems from quantum computers in the near future. The National Institute of Standards and Technology (NIST) is organizing the standardization of PQC algorithms, particularly those for key encapsulation and digital signatures. One candidate selected by NIST in the third round of the standardization process is the lattice-based CRYSTALS-Dilithium digital signature algorithm. We explore the integration of CRYSTALS-Dilithium in a Response-based Cryptography (RBC) protocol to enable quantum resistance. RBC utilizes un-corrected responses from Physically Unclonable Functions (PUFs) as seeds to generate cryptographic keys used for authentication between a server and client device. Authentication is achieved when the server generates a seed from its initially recorded PUF challenge that exactly matches the seed generated from the client device's PUF response. However, there is noise inherent to PUF technology that causes the client's response to differ from the seed recorded on the server during enrollment. The RBC protocol addresses this problem by having the server independently correct its own seed. But, the computational requirements for seed correction increase exponentially with the error rate of the PUF. Therefore, architectures such as Graphics Processing Units (GPUs) are utilized to perform this seed correction in parallel. We propose the first known CRYSTALS-Dilithium implementation on the GPU and use this implementation to develop the first reported Post-Quantum RBC protocol in the literature. We compare our GPU-Accelerated CRYSTALS-Dilithium RBC algorithm to a baseline implementation parallelized using a multi-core CPU. We find that our approach using the GPU achieves speedups of 69.03×, 82.52×, and 90.70× over the CPU for security levels 2, 3, and 5, respectively. To further accelerate the seed correction procedure, we fragment the PUF seed into sub-seeds which allows for a higher error-rate in the PUF given a fixed timing threshold.

* 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.

Keywords: CRYSTALS-Dilithium, Post-Quantum Cryptography, Response-based Cryptography, Physical Unclonable Functions, GPGPU

1 Introduction

Recently, there has been a surge of progress in the development of quantum computers predicted to break the security of public-key algorithms [21]. This vulnerability has led to research efforts in investigating post-quantum cryptography (PQC), a term referring to algorithms that are resistant (in varying degrees) to attacks by quantum computers. The National Institute of Standards and Technology (NIST) is organizing the standardization of PQC algorithms for key encapsulation and digital signatures schemes. One candidate algorithm that has been selected by NIST in the third round of the standardization process is the lattice-based CRYSTALS-Dilithium [9] digital signature algorithm (DSA). With the advent of quantum computers, integration of CRYSTALS-Dilithium and other PQC DSA algorithms to many currently employed protocols, such as Transport Layer Security for Internet communication, looms on the horizon [23,15].

Many critical systems, requiring higher levels of security, incorporate communication from low-powered client devices. However, these low-powered devices often do not have the capability of commissioning higher sercurity levels. Therefore, to enable more secure systems with low-powered devices, new security protocols are needed. The pioneering work of Cambou et al. [6,3] addresses this problem with the response-based cryptography (RBC) protocol.

RBC uses Physically Unclonable Functions (PUFs) to secure communication between a server and client device. For each client device, PUFs offer unique hardware identification [22,12,20,13,14,11,6] and the ability to generate a nearly infinite number of challenge-response pairs for producing cryptographic keys stored only in volatile memory. The RBC protocol works in two phases: enrollment and authentication. During enrollment, a server records a challenge-response pair of a PUF for a client device. During authentication, the client uses a challenge sent by the server to produce a response used as a seed to generate a public key, and sends this public key to the server. The server then uses its initally recorded PUF response to find a matching public key for authenticating the client. But, the inherent noise of PUF technology means that the response of the client device often differs from the initial record on the server [7]. Consequently, the server performs error correction on the seed from its initially recorded PUF response [6,3]. However, conducting error correction on the server is a brute-force search that increases exponentially with the error rate of the PUF.

General purpose computing on graphics processing units (GPGPU) is an attractive alternative to using multi-core CPUs for high throughput search problems. The proposed RBC scheme requires searching large key spaces on a server, and this problem is well-suited to the massively parallel architecture afforded by the GPU. Utilizing a single instruction multiple thread execution model, the GPU lends itself well to RBC since the search space can be partitioned among

threads for independent execution. In this paper, we integrate CRYSTALS-Dilithium with the RBC protocol to enable quantum resistance, and we address the large problem size of error correction by adapting and optimizing the CRYSTALS-Dilithium algorithm on the GPU. In summary, this paper makes the following contributions:

- We present the first reported Post-Quantum RBC algorithm using CRYSTALS-Dilithium on a multi-core CPU. We use this implementation as a reference for comparing against our GPU-accelerated algorithm.
- We develop the first known implementation of CRYSTALS-Dilithium on the GPU.
- We present the first GPU-accelerated CRYSTAL-Dilithium RBC algorithm, CREG. We execute the algorithm on up to four Nvidia V100 GPUs and achieve exceptional multi-GPU scalability. We compare CREG to the CPU reference implementation and demonstrate substantial gains in authentication speed.
- We propose an adaptation of the key generation procedure of CRYSTALS-Dilithium that optimizes CREG by expanding the matrix of the learning with errors instance only once, negating the need of expansion at every iteration of the search.
- Error correction performed by the server incurs a computational cost which limits environments with insufficient capabilities. To accelerate the seed correction procedure and enable an RBC scheme with higher accessibility, we fragment the PUF response seed into sub-seeds which allows for a higher error-rate in the PUF given a fixed timing threshold.

The organization of this paper is as follows. Section 2 outlines related work, Section 3 introduces our GPU-accelerated Post-Quantum RBC algorithm and optimizations, Section 4 presents our experimental evaluation, and finally, Section 5 concludes the paper.

2 Background

2.1 Response-Based Cryptography (RBC)

Discussion of the RBC protocol uses the following terminology. **Client:** Low-powered device of a server-incorported network; **Server:** Computer with computational capacity much greater than the client; **Opponent:** Entity trying to recover the client's PUF response or other private information.

• S_c**:** Client's PUF response used as a seed for generating keys.
• Pk_c**:** Public key generated by the client using its seed, S_c.
• S_s**:** Server's seed generated from the client device's intially recorded PUF challenge.
• Pk_s**:** Public key generated by the server using its seed, S_s.

We outline the RBC engine ran on the server below. We refer the reader to Cambou et. al [5] for a more in-depth description of this engine.

1. A client produces a PUF response from a server-sent PUF challenge and uses this response as a seed to generate public key, Pk_c. The client sends Pk_c to the server.
2. The server uses its intially recorded PUF response as a seed, S_s, to generate public key, Pk_s.
3. The server compares the client's public key, Pk_c, to its public key, Pk_s.
4. If the two public keys match (Pk_s=Pk_c), then the seeds used to generate them are identical and the user is authenticated.
5. If the public keys are different, then the server must find the correct seed using S_s as the starting seed.

If the server's check from above yields unmatching public keys, then a search for a matching public key to the one sent by the client is conducted as follows.

1. The search begins from the initially recorded PUF response seed, S_s.
2. The server iterates over all seeds with a Hamming distance of 1 from S_s. At each iteration, an intermediate seed is used to generate a public key (Pk_s) which is checked for a match with the public key received from the client, Pk_c.
3. If there is a match where the public keys are equal (Pk_s=Pk_c), the seed has been found and the search is haulted.
4. If no match has been found, the Hamming distance is increased by 1, and the search is started again at Step 2.

A server authenticates a client device when a PUF seed, S_s, that generates a public key matching Pk_c is found within a given time constraint, T. This timing constraint is set in such a way to reduce false rejection rates (T is set too low) while also reducing false acceptance rates (T is too high). The knowledge of S_s being close to S_c gives the server an advantage over an opponent without this starting position. The expectation of an opponent's search space cardinality is as follows.

$$n_o = \frac{1}{2} \sum_{j=0}^{256} \binom{256}{j}. \tag{1}$$

Alternatively, the expectation of the server's search space cardinality amounts to the following.

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

We concentrate on the RBC search of size n_s conducted by the server. We implement the algorithm entirely in software. For the purposes of evaluating and constraining the search on the server, we select seeds with a given Hamming distance from the server's intitially recorded response seed. We begin the search from S_s increasing the Hamming distance until S_c is found or a timing threshold, T, has been exceeded. At which point, the client must reattempt authentication.

2.2 CRYSTALS-Dilithium

We note preliminary terminology used in describing CRYSTALS-Dilithium. For consistency, we employ the standard notation described by Ducas et al. [9]. The polynomial ring, $\mathbb{Z}_q[X]/\langle X^n + 1\rangle$, for prime $q = 8380417$, is denoted as R_q. Matrices and vectors of polynomials in R_q are expressed as bold-faced capital and lowercase letters, respectively. Define $S_\eta = \{\mathbf{a} \in R_q \mid \|\mathbf{a}\|_\infty \leq \eta\}$, where $\eta \in \mathbb{N}$.

The Dilithium digital signature algorithm follows the "Fiat-Shamir with Aborts" framework [19] while adopting a variation described by Bai et. al [2]. The security of Dilithium is based on the module learning with errors (MLWE) and module short integer solutions (MSIS) problems [18]. Three security levels numbered 2, 3, and 5 are offered by Ducas et. al [9] in their third round submission to NIST's PQC competition. These different security levels correspond to increasing and decreasing the pair (k, l) which determines the size of the matrices used in the algorithm. There are three procedures which consitute the entirety of the algorithm: key generation, signing, and verification. These procedures are computationally bounded by two operations: multiplication in R_q via the number theoretic transform (NTT) and hashing via an extendable output function (XOF).

Since the search procedure of the RBC protocol only involves generating a public key, we focus solely on Dilithium's key generation procedure which we outline as follows.

Algorithm 1 Dilithium Key Generation

1: **procedure** keyGeneration
2: $r_1 \leftarrow \{0,1\}^{256}$
3: $r_2 \leftarrow \{0,1\}^{256}$
4: $\mathbf{A} \leftarrow \text{expandA}(r_1)$
5: $(\mathbf{s_1}, \mathbf{s_2}) \leftarrow \text{genSecretVectors}(r_2)$
6: $\mathbf{p} \leftarrow \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$
7: $(\mathbf{p_1}, \mathbf{p_0}) \leftarrow \text{decompose}(\mathbf{p})$
8: $d \leftarrow \text{H}(r_1 \mid\mid \mathbf{p_1})$
9: **return** $(pk = (r_1, \mathbf{p_1}), sk = (r_1, r_2, d, \mathbf{s_1}, \mathbf{s_2}, \mathbf{p_0}))$

The key generation procedure begins on line 2 and line 3 by generating two 256-bit random seeds. Line 4 expands r_1 to create the public parameter, $\mathbf{A} \in R_q^{k\times l}$. Then, line 5 samples small secret vectors of polynomials, $\mathbf{s_1} \in S_\eta^l$ and $\mathbf{s_2} \in S_\eta^k$. Using these secret vectors, line 34 creates the MLWE instance, $\mathbf{p} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$. Line 7 then decomposes $\mathbf{p}$ into $(\mathbf{p_1}, \mathbf{p_0})$ where $\mathbf{p_1}$ consists of the higher order bits and $\mathbf{p_0}$ consists of the lower order bits. Then, on line 8, r_1 is hashed using SHAKE-256 with $\mathbf{p_1}$ to produce d. Finally, line 9 publishes the public key as $(r_1, \mathbf{A})$ and the secret key as $(r_1, r_2, d, \mathbf{s_1}, \mathbf{s_2}, \text{and } \mathbf{p_0})$.

2.3 CRYSTALS-Dilithium on the GPU

To the best of our knowledge, no prior work on fully adapting the CRYSTALS-Dilithium algorithm to the GPU has been published. However, there are published works that propose GPU-based optimizations for the NTT used in Dilithium [10,1]. All reported optimizations of the NTT split the work of a single tranform amongst multiple threads. Since we find that the optimal parallel granularity is one thread per key generation, we depart from these optimizations and instead use the reference implementation of NTT provided by Ducas et al. [9].

3 CREG: CRYSTALS-Dilithium Response-Based Cryptography Engine Using GPGPU

3.1 Noise Injection

In practice, the RBC protocol utilizes hardware PUF responses for seeds to generate cryptographic keys. In contrast, we perform all operations in software by randomly generating a client device's PUF response seed, S_c, of length 256 bits. To simulate inherent noise that would occur in a hardware PUF, we flip d bits of S_c to produce the server's PUF seed, S_s. This implies that S_s is a Hamming distance d away from S_c. We evaluate different Hamming distances in our experimental evaluation striving for the highest noise level which also provides authentication within a reasonable timing threshold.

3.2 Parallelizing the Error Correction Workload

Assume the server's 256-bit PUF seed is some Hamming distance, $d>0$, from the client's PUF seed. Let $j \in \mathbb{N}$, such that $1 \leq j \leq d$. Then, we define a set, P_j, that consists of every 256-bit string that is exactly a Hamming distance j from S_s. The cardinality of every P_j is as follows.

$$|P_j| = \binom{256}{j}. \tag{3}$$

Every P_j will be iterated by t_j number of threads launched by a GPU kernel invocation. For a given Hamming distance, j, we define the workload of each thread to be the following.

$$n_j = \frac{|P_j|}{t}. \tag{4}$$

For the sake of brevity, and without loss of generality, we assume that $|P_j| \bmod t{=}0$ (if $|P_j| \bmod t \neq 0$, then we distribute the remaining work to threads to be processed). We enumerate each set, P_j, as described in Knuth [17]. A thread's starting and ending position is determined by their thread id and the parameter n_j. Iterating across its own workload, a thread maps the current ordinal number to a 256-bit string (being a Hamming distance j from S_c) by using *Gosper's hack*, as demonstrated in Knuth [16]. In summary, the workload induced by each Hamming distance is partitioned into disjoint batches based on the enumeration scheme, allowing each thread to independently iterate over its own batch.

3.3 Early Termination Strategy

Assuming the distribution of error bits in a PUF response seed is uniform, the server will find a client-matching PUF seed in P_j at random. Therefore, the server does not need to iterate and examine all of P_j. Rather, once a matching public key has been found, the search should be terminated. To achieve early termination, CREG stores a flag in unified memory shared amongst all threads. If a thread finds a match, the flag is updated. Threads check every r iterations of their own search whether the flag has been set. A set flag causes a thread to stop its search early and return to the host.

3.4 Fragmentation

Error correction conducted by the server is a brute-force search that increases exponentially with the error rate of the PUF, as expressed in Equation 2. Therefore, even an RBC engine that employs a computing cluster with significant computational resources may find a Hamming distance of $d>5$ to be intractable. To address this problem, we reduce the search space defined in Equation 2, by implementing a fragmentation scheme outlined by Cambou et al. [3]. In this scheme, the client and server fragment their respective 256-bit PUF seeds into g subseeds of size $\frac{256}{g}$. Then, an error-free padding of size $256 - \frac{256}{g}$ is concatenated to the subseeds using a client-server shared scheme called Ternary Addressable Public Key Infrastructure [4]. These padded subseeds are then used to produce g public keys. The server searches only the non-padded portions of the PUF seeds. Without fragmentation, the upper bound of the search space is $\sum_{j=0}^{d} \binom{256}{j}$. Employing fragmentation reduces the upper bound of the server's workload to the following:

$$n_s(d, g) = (g - 1) + \sum_{j=0}^{d} \binom{256/g}{j}. \tag{5}$$

Fragmentation reduces the error correction workload requirements. This allows PUFs having high(er) error rates to be able to employ the RBC scheme, thus increasing the practicality of PUF technology.

3.5 Optimizing Dilithium Key Generation

The Dilithium algorithm (Section 2.2) is reported to be computationally bounded by two operations: expansion of an XOF via SHAKE-256 and multiplication in the polynomial ring, R_q [9]. We focus on removing calls to the former operation for optimization.

Removing matrix expansion: During key generation, the procedure creates an expanded matrix A with a call to an XOF via SHAKE-256. We remove this expansion operation. To achieve this, we extract r_1 from the public key sent by the client, and use r_1 to create A on the host only once which we then transfer to global memory on the GPU. Matrix A is shared amongst all threads conducting the RBC search.

Removing collision resistant hash: The Dilithium key generation procedure squeezes a random seed, r_1, and small vector of polynomials, p_1, into a collision resistant hash digest via SHAKE-256. This hash digest is part of the packed secret key. Since the RBC search does not require use of the packed secret key, we remove this call to the collision resistant hash function.

3.6 Algorithm Overview

Algorithm 2 outlines CREG, which accelerates RBC by executing an enumeration scheme for iterating large workloads, deploying an early exit strategy, optimizing Dilithium key generation, and utilizing multiple GPUs.

Algorithm 2 begins on line 2 by initializing a variable (seedMatch) for counting the number of matching subseeds found. The client is authenticated if the value of seedMatch is equal to the input fragmentation parameter, g. Then, line 3 instantiates a loop across the g subseeds (if g=1, then there is no fragmentation employed). Line 4 then generates the client's i^{th} subseed using parameters S_s, d, g, and i. Line 5 initializes a variable, subSeedFound, for keeping track of a recovered subseed. Using the client's i^{th} public key, line 6 generates the matrix, A. Then, the intermediate public key, Pk_s, is generated on line 36 using the ith subseed, S_i, and matrix A. This public key is checked for equality against the client's i^{th} public key and the result is added to the seedMatch variable (line 8). Looping across an increasing Hamming distance, j, until reaching d, begins on line 10 given that the public keys, Pk_s and $Pk_c[i]$, are unequal (line 9). Then, the number of seeds to compute ($|P_j|$), the number of threads to conduct the search (t), and the workload of each thread (n_j) are computed on line 11, line 12, and line 13, respectively. A unified memory variable (unifiedCheck) shared amongst all threads of a GPU is then initialized in the case that an early termination procedure is conducted (line 28). Line 15 then launches a GPU kernel with parameters, S_i, subSeedFound, unifiedCheck, j, $Pk_c[i]$, t, n_j, and A. The resulting value of subSeedFound from the GPU kernel is added to the seedMatch variable (line 16). Finally, line 17 returns the logical equivalence comparison of seedMatch to the fragmentation level, g.

The procedure, GpuKernel (line 18), is outlined as follows. Line 19 and line 20 gather the thread's id (tid) and initialize an intermediate public key (Pk_s), respectively. Then, line 21 generates the pair of starting and ending bit strings for deliminating a thread's workload space. Line 22 then uses this starting and ending pair along with the seed, S_i, to generate an iterator, currSeed. Then, line 23 instantiates a loop that continues until either the iterator reaches an end (endSeed) or the seed has been found and a variable, earlyExit, is activated for early termination. The intermediate public key, Pk_s, is then generated using the iterator, currSeed, and matrix A (line 24). Then, line 25 compares this public key to the client's public key (Pk_c). If the public keys are equal, then line 26 sets the subSeedFound variable to true. Then, line 28 checks if the global directive flag (earlyExit) for early termination is set. If conducting an early termination procedure, then line 14 sets the unified memory variable, unifiedCheck, to true for directing all threads to terminate their search early. Line 29 then updates

Algorithm 2 CREG Algorithm

```
1: procedure CREG(Pk_c[g], S_s, d, g)
2:     seedMatch ← 0
3:     for i ∈ 0,...,g − 1 do
4:         S_i ← genFragSeed(S_s, d, g, i)
5:         subSeedFound ← False
6:         A_i ← genMatrix(Pk_c[i − 1])
7:         Pk_s ← genPublicKey(S_i, A_i)
8:         seedMatch = seedMatch + (Pk_s = Pk_c[i])
9:         if Pk_s ≠ Pk_c[i] then
10:            for j ∈ 1,...,d do
11:                |P_j| ← computeNumSeeds(d)
12:                t ← computeNumThreads(|P_j|)
13:                n_j ← computeSeedsPerThread(|P_j|, t)
14:                unifiedCheck ← False
15:                GpuKernel(S_i, subSeedFound, unifiedCheck, j, Pk_c[i], t, n_j, A_i)
16:                seedMatch = seedMatch + subSeedFound
17:    return (seedMatch = g)

18: procedure GpuKernel(S_i, subSeedFound, unifiedCheck, j, Pk_c, t, n_j, A)
19:    tid ← getThreadId()
20:    Pk_s ← ∅
21:    (startComb, endComb) ← getCombPair(tid, j, n_j)
22:    (currSeed, endSeed) ← initIterator(S_i, startComb, endComb)
23:    while !(earlyExit and unifiedCheck) and (currSeed ≠ endSeed) do
24:        GenPublicKey(Pk_s, currSeed, A)
25:        if Pk_c = Pk_s then
26:            subSeedFound ← True
27:            if earlyExit then
28:                unifiedCheck ← True
29:        currSeed ← getNextSeed()
30:    return

31: procedure GenPublicKey(Pk_s, seed, A)
32:    (f_1, f_2) ∈ {0,1}^512 ← XOF(seed)
33:    (s_1, s_2) ← genSecretVectors(f_2)
34:    t ← As_1 + s_2
35:    (t_1, t_0) ← decompose(t)
36:    Pk_s = t_1
37:    return
```

the iterator to obtain the next seed. If a match between public keys (line 25) is never found, then the subSeedFound variable is never updated and the kernel exits.

The procedure, GenPublicKey, is outlined as follows (line 31). Line 32 begins by hashing the PUF seed into a digest of length 512 bits (referenced by f_1 and f_2). Then, line 33 generates the secret short vectors s_1 and s_2 using the

Table 1. Platform details. All platforms use Intel Xeon processors.

Platform	Model	# Nodes	Cores (Total)	Clock	Memory
PLAT1	1×W-2295 (Cascade Lake)	1	1×18	3.0 GHz	256 GiB
PLAT2	2×Gold 6132 (Skylake)	1	2×14 (28)	2.6 GHz	384 GiB

Table 2. GPU platform details, using only Nvidia GPUs.

Platform	Model	Cores/GPU	Memory/GPU	Software
PLAT2	4×Tesla V100	5120	16 GiB	CUDA 11.2

last 256 bits of our previous hash digest, f_2. Note that the first 256 bits of our hash digest, referenced by f_1, is not used in this procedure since it is only used to create the input parameter A. Line 34 then multiplies A by the short vector s_1 and adds short vector s_2 to produce t. As explained in Section 2.2, the high order bits of t are extracted to produce t_1 and t_0 (line 35). Finally, line 36 saves t_1 as the public key, Pk_s, and the procedure exits.

To accommodate multiple GPUs, Algorithm 2 is reconfigured as follows. An authentication variable, subSeedFound, is initialized for each GPU device on line 5. Next, variables $|P_j|$ and n_j initialized on lines 11 and 13 are evenly distributed to kernel invocations for each device (line 15). Finally, line 16 is changed by adding to seedMatch all subSeedFound variables of each device.

4 Experimental Evaluation

4.1 Experimental Methodology

We employ two platforms that are detailed in Tables 1 and 2. PLAT1 is equipped with 18 Intel Xeon CPU cores of the Cascade Lake architecture. PLAT2 is equipped with four Volta GPUs (V100). We use PLAT1 for running our multi-core CPU OpenMP implementation, as the CPU is of a newer generation than that in PLAT2. We use PLAT2 to showcase the high performing capabilities of our algorithm as executed on multiple GPUs within a single computer. Being part of our institution's cluster, PLAT2 is shared by multiple users, and we use dedicated resources.

All CPU code is written in C/C++ and is compiled using the GNU compiler with the O3 optimization flag. All GPU code is written in CUDA. Unless otherwise stated, in all experiments we average our results over 5 time trials. We outline the configurations of the implementations that we compare below.

CREG: Recall from Section 3.2 that n_j is the workload of each GPU thread and from Section 3.3 that r is number iterations before checking if the public key has been found. In all experiments on PLAT2, we set n_j=16, r=1, and the number of threads per CUDA block to be 32, since these values experimentally yield good performance.

CPU-OMP: We employ a similar procedure to the one outlined in Section 3. In place of threads launched from a GPU kernel invocation, we launch threads via OpenMP on a multi-core CPU and use this as our reference implementation for comparing to our GPU-accelerated algorithm, CREG.

4.2 Evaluation of CREG

We organize our experiments as follows. We begin by determining a good workload distribution parameter, n_j. We evaluate CREG by conducting exhaustive searches and early termination searches. We compare exhaustive searches to our reference implementation, CPU-OMP. We demonstrate the scalability of CREG using multiple GPUs through our early termination searches. Finally, we show and discuss the capabilities of CREG when employing fragmentation.

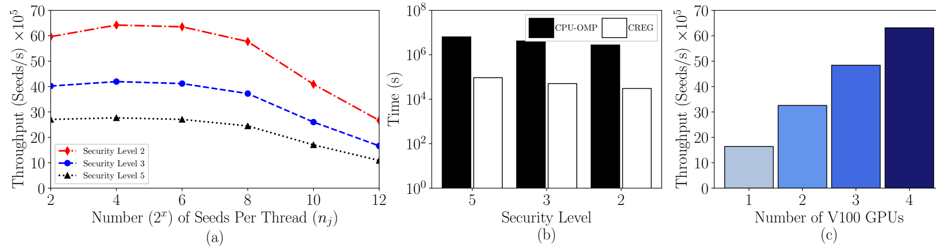

Fig. 1. Three plots are shown as follows: (a) throughput in seeds/s vs. the workload of each thread (n_j), comparing the different Dilithium security levels for an exhaustive search on PLAT2 with d=4, and g=1; (b) execution time (in seconds) plotted on a log scale for CREG on PLAT2 and CPU-OMP on PLAT1, comparing the different Dilithium security levels for an exhaustive search with d=4, and g=1; and (c) throughput in seeds searched per second for CREG on PLAT2, comparing an increasing number of GPUs using only Dilithium security level 2 for an early termination search with d=4, and g=1.

Determining the Workload of Threads: The workload (number of seeds to be searched) of each thread is defined by the size of n_j. Varying n_j corresponds to increasing or decreasing the number of threads for performing the RBC search. Figure 1(a) plots the seed search throughput in seeds/s vs. the number of seeds assigned to each thread (n_j) for Dilithium security levels 2, 3, and 5 on PLAT2 for an exhaustive search with g=1 and d=4. We observe a local throughput maximum for each security level occurring at the same value of n_j=2^4.

Exhaustive Search: To demonstrate performance gains achieved through GPU acceleration, we conduct an exhaustive search with CREG and CPU-OMP. Figure 1(b) plots the seed search throughput in seeds/s for CREG and CPU-OMP on PLAT2 and PLAT1, respectively, with an exhaustive search using d=4 and g=1. We find that CREG achieves speedups of 69.03×, 82.52×, and 90.70× over CPU-OMP for security levels 2, 3, and 5, respectively.

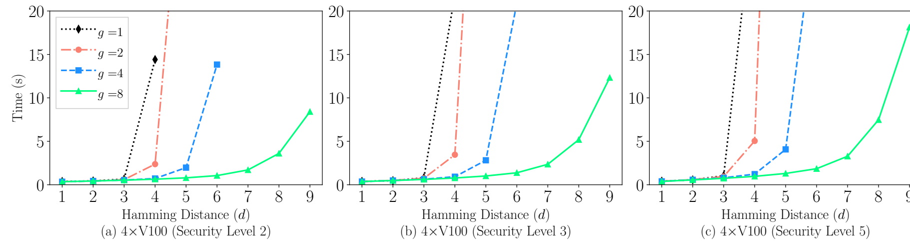

Fig. 2. Response time vs. Hamming distance for selected fragmentation levels, g, for Dilithium security levels 2, 3, and 5 corresponding to plots (a), (b), and (c), respectively. For g=1 the average case is conducted using Equation 2, and for all g>1, an exhaustive search is carried out using our upper bound from Equation 5.

Early Exit: We evaluate our algorithm, CREG, with an early termination procedure (Section 3.3) and simultaneously show the scalability of CREG to multiple GPUs. Figure 1(c) plots the throughput in seeds searched per second using CREG with 1, 2, 3, and 4 V100 GPUs with an early termination search using Dilithium security level 2 and g=1. From 1xV100 to 4xV100 we observe above 95% parallel efficiency.

Fragmentation: Figure 2 plots the response time vs. Hamming distance on PLAT2 for selected fragmentation levels, g, and Dilithium security levels 2, 3, and 5 where the average number of seeds to search, $n_s(d)$, is used when g=1, and $n_s(d, g)$ is used when g>1. We observe that when g=8, $d \leq 8$ is tractable for Dilithium security level 2 and $d \leq 7$ is tractable for Dilithium security levels 3 and 5 within a timing threshold of T=5 s. Employing fragmentation enables higher levels of error rates in PUF response seeds to be tractable within a reasonable timing threshold, making RBC more accessible to a larger class of computing platforms.

5 Discussion and Conclusions

We have presented CREG, a CRYSTALS-Dilithium response-based cryptography engine using GPGPU. The engine enables secure client-server communications using PUF technology. Inherent noise in PUFs makes the client's public key differ from the public key generated on the server using the client's initially recorded challenge. To our knowledge, this paper presents the first known implementation of CRYSTALS-Dilithium on the GPU and uses this implementation to present the first reported Post-Quantum RBC engine. We evaluated CREG on 4xV100 GPUs and observe superior error-correction throughput compared to a CPU-based implementation while also reporting high parallel efficiency using multiple GPUs. The implementation of Dilithium on the GPU presented in this paper may be useful to other researchers interested in using a batched execution. Future work includes extending the RBC protocol to other PQC algorithms, including a key encapsulation scheme, such as SABER [8].

References

1. Akleylek, S., Tok, Z.Y.: Efficient arithmetic for lattice-based cryptography on gpu using the cuda platform. 2014 22nd Signal Processing and Communications Applications Conference (SIU) pp. 854–857 (2014)
2. Bai, S., Galbraith, S.: An improved compression technique for signatures based on learning with errors. IACR Cryptol. ePrint Arch. **2013**, 838 (2014)
3. Cambou, B.: Unequally powered cryptography with physical unclonable functions for networks of internet of things terminals. In: 2019 Spring Simulation Conference (SpringSim). pp. 1–13 (2019)
4. Cambou, B., Telesca, D.: Ternary computing to strengthen information assurance. development of ternary state based public key exchange. In: IEEE, SAI-2018, Computing Conference (2018)
5. Cambou, B., Gowanlock, M., Heynssens, J., Jain, S., Philabaum, C., Booher, D., Burke, I., Garrard, J., Telesca, D., Njilla, L.: Securing additive manufacturing with blockchains and distributed physically unclonable functions. Cryptography **4**, 17 (06 2020). https://doi.org/10.3390/cryptography4020017
6. Cambou, B., Philabaum, C., Booher, D., Telesca, D.A.: Response-based cryptographic methods with ternary physical unclonable functions. In: Future of Information and Communication Conference. pp. 781–800. Springer (2019)
7. Chen, S., Li, B., Cao, Y.: Intrinsic physical unclonable function (puf) sensors in commodity devices. Sensors **19**(11), 2428 (2019)
8. D'Anvers, J.P., Karmakar, A., Roy, S., Vercauteren, F.: Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. In: IACR Cryptol. ePrint Arch. (2018)
9. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-dilithium algorithm specifications and supporting documentation (2017)
10. Emeliyanenko, P.: Efficient multiplication of polynomials on graphics hardware. In: APPT (2009)
11. Gao, Y., Ranasinghe, D.C., Al-Sarawi, S.F., Kavehei, O., Abbott, D.: Emerging physical unclonable functions with nanotechnology. IEEE access **4**, 61–80 (2016)
12. Gassend, B., Clarke, D., Van Dijk, M., Devadas, S.: Silicon physical random functions. In: Proceedings of the 9th ACM conference on Computer and communications security. pp. 148–160. ACM (2002)
13. Herder, C., Yu, M.D., Koushanfar, F., Devadas, S.: Physical unclonable functions and applications: A tutorial. Proceedings of the IEEE **102**(8), 1126–1141 (2014)
14. Jin, Y.: Introduction to hardware security. Electronics **4**(4), 763–784 (2015)
15. Kampanakis, P., Sikeridis, D.: Two post-quantum signature use-cases: Non-issues, challenges and potential solutions (11 2019)
16. Knuth, D.E.: The Art of Computer Programming, vol. 4. Addison Wesley Professional (2009)
17. Knuth, D.E.: Generating all combinations and partitions. Addison Wesley Professional (2010)
18. Langlois, A., Stehle, D.: Worst-case to average-case reductions for module lattices. Cryptology ePrint Archive, Report 2012/090 (2012), https://eprint.iacr.org/2012/090
19. Lyubashevsky, V.: Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In: ASIACRYPT (2009)

20. Maes, R., Verbauwhede, I.: Physically unclonable functions: A study on the state of the art and future research directions. In: Towards Hardware-Intrinsic Security, pp. 3–37. Springer (2010)
21. Mavroeidis, V., Vishi, K., Zych, M., Jøsang, A.: The impact of quantum computing on present cryptography. ArXiv **abs/1804.00200** (2018)
22. Pappu, R., Recht, B., Taylor, J., Gershenfeld, N.: Physical one-way functions. Science **297**(5589), 2026–2030 (2002)
23. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Post-quantum authentication in tls 1.3: A performance study. IACR Cryptol. ePrint Arch. **2020**, 71 (2020)