A Symmetric Cipher Response-Based Cryptography Engine Accelerated Using GPGPU

Jordan Wright*, Zane Fink†, Michael Gowanlock*, Christopher Philabaum*, Brian Donnelly*, Bertrand Cambou*
*School of Informatics, Computing, & Cyber Systems, Northern Arizona University, Flagstaff, AZ, USA
†Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
Email: jaw566@nau.edu, zanef2@illinois.edu, michael.gowanlock@nau.edu,
cp723@nau.edu, bwd29@nau.edu, bertrand.cambou@nau.edu

***Abstract*—Many low-powered devices, such as those in the Internet of Things (IoT), require high levels of security. One shortfall of cryptographic systems is the storage of private key information in non-volatile memory that an opponent can read. Client devices can generate private keys on-demand using a physically unclonable function (PUF) to obviate this problem. However, low-powered devices may not have the computational resources to correct for the error in the PUF relative to the initially recorded PUF challenge. Response-based cryptography (RBC), when combined with encrypting schemes such as the Advanced Encryption Standard (AES), addresses this problem by having a secure server perform a search over the key space starting from a client device's initially recorded challenge. We propose an RBC engine based on symmetric ciphers that uses graphics processing units (GPUs). We use the GPU to perform a massively parallel search over the key space to authenticate the client's key(s). The computational requirements for executing the search and authenticating the user within a time threshold, T, increase exponentially. This limits the classes of computers that are able to perform the search. To address this problem, we employ a scheme that generates subkeys from the PUF. This increases the granularity of computational capabilities that are able to perform the RBC search within the selected T=5 s authentication threshold. We compare our algorithm, GRBC, to an OpenSSL-based MPI reference implementation executed on up to 512 CPU cores. Our approach using the GPU achieves superior key search throughput over the CPU.**

***Index Terms*—AES, SPECK, ChaCha20, GPGPU, Physical Unclonable Function, Response-based Cryptography**

I. Introduction

Client devices that communicate via insecure networks, such as the Internet, are vulnerable to cyberattacks. Many client devices, such as those in the Internet of Things (IoT), are low-powered and do not have the capacity to realize might levels of security. New security architectures are needed to reduce the burden of computation on low-powered devices. In this paper, we propose accelerating the response-based cryptography (RBC) [1], [2] architecture for the authentication of low-powered devices using the massive parallelism afforded by general-purpose computing on graphics processing units (GPGPU).

To authenticate a device, Physically Unclonable Functions (PUFs) can generate keys in hardware on-demand. Due to variations in manufacturing, each client device will have its own unique PUF "fingerprint" in hardware [1], [3], [4] which guarantees the authenticity of the device. PUFs generate a massive number of possible keys and mitigate against storing keys in non-volatile memory that can be read by an opponent. PUFs age and are susceptible to environmental effects, such as temperature [5]. Thus, keys generated by a PUF can change over time. Since cryptographic schemes cannot tolerate any error in a key, PUF technology needs to be augmented by other means such as data helpers and error correction schemes [6]–[8].

Due to the necessity of client error correction, PUF technology may be unsuitable for IoT devices with low computational resources. Response-based cryptography addresses this problem [1], [2]. Instead of client-side error correction, a server with high computational throughput is used to find the correct private key (response) using the client's initially recorded "fingerprint" as a reference with a known challenge. Thus, RBC enables low-powered IoT devices to use PUF technology. The image of the PUF, which contains multiple fingerprints for each client device, is initially recorded and stored on a server during a secure enrollment process. Once the client device is deployed, the PUF challenges and reference responses stored on the server are used to generate cryptographic keys to authenticate and securely communicate with a client; the responses from the client's PUF are used to generate cryptographic keys closely matching the ones concurrently generated by the server.

In this paper, we exploit the RBC architecture by assuming the client's response, used as a private key for symmetric encryption contains errors, and is thus some Hamming distance away from the server's reference response/challenge pair. Then, the server must perform a search, starting from the server's challenge, to uncover the client's private key. Since the cost of the server-side key search is high through a brute force attack, an opponent trying to uncover the client's private key without knowing the image of the PUF would not have the computational resources needed to gain this information.

In RBC, since the search space for the private key is

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.

large and can be carried out in parallel by multiple cores, the search is highly amenable to Graphics Processing Unit (GPU) acceleration. We propose an RBC algorithm based on symmetric ciphers that is parallelized using the GPU. We find that our algorithm is highly efficient and outperforms a multi-core CPU reference implementation. This paper makes the following contributions:

- We present the first GPU-accelerated RBC algorithm, GRBC. We adapt the algorithm for symmetric ciphers AES, SPECK, and ChaCha20. We execute the algorithm on 3 platforms and on up to four Nvidia V100 GPUs and achieve high multi-GPU scalability. GRBC achieves substantial performance gains over the CPU reference implementation.
- We propose a GPU algorithm optimization that includes generating the keys used in each round of AES on-demand, negating the need to store the entire key in GPU memory.
- Response-based cryptography requires sufficient computational resources to carry out the search within a reasonable authentication time threshold. We propose fragmenting the PUF-generated key into several subkeys. This increases the granularity of computational capabilities that can carry out the key search, thus making RBC deployable in a wider range of environments within our prescribed T=5 s authentication time threshold. GRBC Is the first RBC algorithm suitable for execution an a single node, thus making it a good candidate for real-world RBC implementations.

The paper is organized as follows. Section II outlines related work, Section III presents our GPU-accelerated RBC algorithm and optimizations, Section IV presents our experimental evaluation, and finally, Section V concludes the paper.

II. Background

A. Terminology

We use the following terminology and notation. **Client:** User equipped with a low powered device to be authenticated by a server; **Server:** A computer with significantly more computational capacity than the client; **Opponent:** A person/entity wishing to recover the client's private key, K_c; **Challenges:** Data streams of any radix to used in querying a PUF; **Responses:** Data stream outputs used as private keys that are generated by querying a PUF with a challenge; **Lookup Table:** A secure database on the server that stores PUF challenge-response pairs for each client device. K_c**:** The client's private key. K_s**:** A private key found in a lookup table on the server as a function of the client's user ID that is similar to the client's private key, K_c. C_c**:** Ciphertext generated by the client using its private key, K_c. C_s**:** Ciphertext generated by the server using the private key, K_s.

B. Response-Based Cryptography (RBC)

RBC requires server-conducted network enrollment of each client prior to authentication. Enrollment occurs only once in a secure environment and results in a lookup table of PUF challenge-response pairs stored on the server. The following overview of the RBC authentication scheme assumes a client has been successfully enrolled by the server. For additional information, see Cambou et al. [1].

Step 1: The server and client conduct a handshake by which the server sends instructions (or challenges) to the client, indicating which addresses within the PUF should be used to generate a private key. Step 2: A client generates its private key, denoted as K_c, using its PUF and server-sent instructions. The client then takes its plain text user ID and encrypts it using its private key, K_c. This creates the ciphertext, C_c. The client sends the server its user ID and ciphertext. Step 3: The server independently uses the instructions it sent to the client and its lookup table to obtain a private key, denoted as K_s. Then, the server encrypts the user ID using K_s. This generates the ciphertext C_s. Step 4: The server compares the cipher received from the client (C_c) to the cipher it generated using the key in its lookup table (C_s). Step 5: If the two ciphers match (C_s=C_c), then the keys are identical, and the user is authenticated. Step 6: If the keys are different, the server needs to find the correct key using K_s as the starting key.

Figure 1 shows an example of the process outlined above. In the figure, the RBC search procedure is not shown. The search procedure uses an iterative process that is outlined as follows: Step 1: The search procedure begins using the server's private key K_s, which is similar but not identical to the client's private key, K_c. Step 2: Using K_s, the search begins by iterating over all keys with a Hamming distance of 1 from K_s, encrypting the user ID and checking if the ciphertext matches the ciphertext received from the client, C_c. Step 3: If there is a match where the ciphertexts are equal (C_s=C_c), the key has been found, and the search stops. Step 4: If there is no match, the Hamming distance increases by 1, and the search begins at Step 2.

Security Considerations: i) The handshake conducted by the server and client mitigates against an opponent's ability to conduct standard replay (or spoofing) attacks since, for each authentication attempt, unique private keys are used corresponding to different ciphertexts; ii) The complexity of the search increases exponentially with the Hamming distance, thus increasing the authentication latency. A user is authenticated if the server's search can find the client's private key, K_c, within a given time constraint, T. Therefore, with a sufficiently large Hamming distance, an opponent with knowledge of the server's starting position, K_s, will be unable to recover K_c, even with access to substantial computational resources. Furthermore, it is assumed that the opponent does not have knowledge of the server's starting position.

In this paper, we focus on the RBC search phase shown in Figure 1. To begin this search, we generate K_c and then add noise to K_c to create K_s. The search starts from this value of K_s, which we then recover using an increasing Hamming distance, d=1, then d=2, and so on. In practice, if T is exceeded, authentication fails, and the user must reattempt authentication.

On average, for a given Hamming distance, d, K_c will be found by the server halfway through the entire key space. The search is terminated once the key is found. The *average case* number of keys searched by the server, denoted as n_s, for

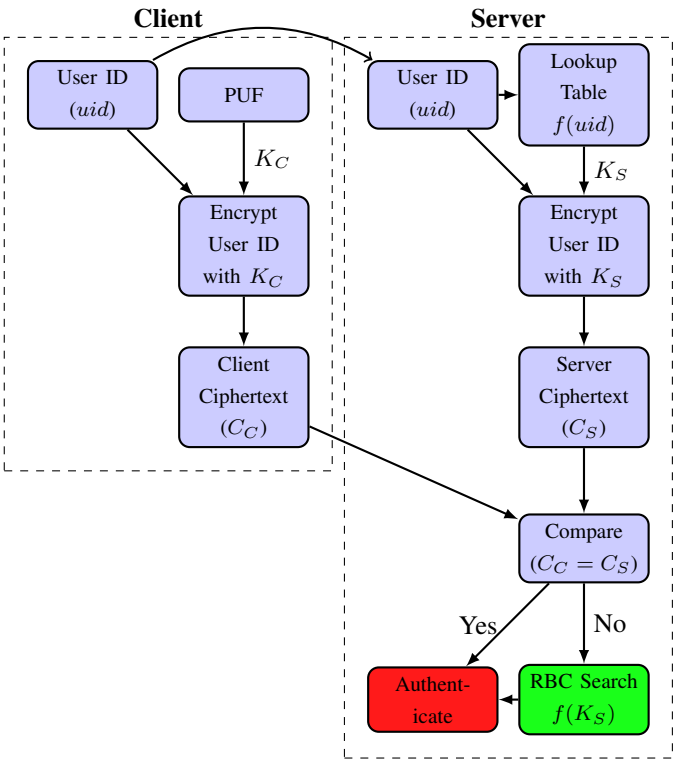


Fig. 1. Illustration of response-based cryptography. See text for details.

Hamming distance, d, is as follows:

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

Consider an opponent without any knowledge of the *starting position* of the server's search (which is K_s). The *worst case* is searching all keys up to d=256. The *average case* requires searching half of the total number of keys. Thus, the average number of keys searched by the opponent, denoted as n_o, is as follows:

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

Consequently, the server's search is constrained because the starting position is known, whereas the opponent's search is exhaustive.

An illustrative example: Consider a 256-bit key, K_c, a Hamming distance, d=5, and $K_s = \text{kgen}(K_c, d)$, which denotes randomly generating the server's key, K_s, from the client's key, K_c, using d randomly flipped bits. Using Equations 1 and 2, we find that the opponent needs to search a factor $\approx 1.26 \times 10^{67}$ of the server's keys. This example shows the disparity in the search space cardinality between the server and the opponent. Without knowledge of K_s, the opponent will be unable to recover the client's private key.

C. GPU-Accelerated Advanced Encryption Standard (AES)

Several papers have proposed AES algorithms for the GPU to maximize encryption throughput [9]–[18]. An example application would be to use AES to encrypt a user's hard drive using their private key.

A pioneering GPU AES paper by Cook et al. [9] proposed an OpenGL-based algorithm. The authors report several drawbacks of GPU architecture for AES, including insufficient API support. However, later papers (described below) leveraging APIs for general-purpose computing (CUDA [19] and OpenCL [20]) were able to overcome many of the obstacles reported in this paper. Biagio et al. [10] proposed optimizations for AES-CTR that include storing lookup tables in shared memory and permuting the data in shared memory to avoid bank conflicts. They achieve a speedup up to 14× over their baseline OpenSSL [21] CPU implementation. Lee et al. [16] proposed optimizations for AES-CTR and exploited the warp shuffle primitive that allows threads in the same warp to read from each other's registers. The optimization improves performance as register accesses are faster than shared memory accesses. Abdelrahman et al. [17] proposed optimizing AES-ECB using shared memory for lookup tables and determined the best execution parameters, including the number of threads per block and encrypted bytes per thread. On an Nvidia GTX 1080, the authors achieve 279.86 Gbps encryption throughput on AES-128. The authors expanded this work to include optimizations for PCIe host/device data transfers and showed that while they achieve high encryption throughput on the GPU, data transfers between host and device are the main bottleneck when encrypting large data volumes [18].

Contrasting GPU AES implementations and RBC requirements: While efficient AES implementations are important for achieving high search throughput for RBC, there are several irrelevant AES optimizations. RBC requires generating a key, encrypting the data to create the ciphertext, and checking if the ciphertext on the server matches the ciphertext received from the client (Figure 1). In contrast to the abovementioned GPU AES works, RBC encrypts across a large key space. Therefore, the total computational load is a combination of key generation and encryption. Also, optimizations that are employed to improve the performance of encrypting a single large volume of data with a private key (e.g., key expansion on the CPU or assignment of work to threads, such as 16–32 bytes/thread [18]) are not useful for RBC since they do not apply across different keys. Also, as discussed in Abdelrahman et al. [18], PCIe data transfers are a bottleneck for AES. However, the search for the correct key in RBC only requires finding the key on the GPU and returning it to the host. Consequently, host/device data transfers over PCIe are not prohibitive for RBC.

D. GPU AES Baseline Implementation

Lin et al. [22] proposed a GPU AES algorithm that is resistant to side-channel attacks. AES is susceptible to side-channel attacks because algorithms typically use lookup tables to improve efficiency (Section II-C), and accesses to lookup

tables may have bank conflicts. Also, accesses to global memory may be un-coalesced. Both bank conflicts and un-coalesced memory accesses have higher access latencies than conflict-free shared-memory accesses and coalesced memory accesses, respectively. This latency information can be exploited to allow an opponent to guess the bytes in a key through this timing channel attack [22], [23]. To address this, Lin et al. [22] propose a scatter-and-gather approach that yields constant memory access latency, thus preventing timing channel attacks.

We depart from the side-channel resistant work of Lin et al. [22] as the GPUs used in our work for RBC are assumed to be located in a secure server. However, the GPU AES implementation by Lin et al. [22], without their side-channel resistant functionality, achieves 289 Gbps on a GTX 1080 (Pascal generation GPU). To our knowledge, this GPU implementation is the fastest available in the literature, and we use it in our RBC algorithm. Their source code is publicly available[1].

Other side-channel attacks can be successful, particularly those targeting key generators with Differential Power Analysis. One of the advantages of our method is that we concurrently execute AES encryption on multiple keys. Therefore, the power needed by the GPU to execute RBC is the combination of concurrent key generation and encryption, thus confusing an opponent. In addition, unlike traditional schemes based on Error-Correcting Codes, the proposed scheme is relying on a server, which is less sensitive to side-channel attacks than client devices located in a network.

E. GPU SPECK and ChaCha20 Baseline Implementations

SPECK is a lightweight block cipher released by the National Security Agency [24]. The cipher was designed to operate well on a diverse collection of IoT devices. Given the lack of literature on GPU implementations of SPECK, we base our CUDA implementation on the specification paper.

ChaCha20 is a stream cipher built as a variation of Salsa20 [25]. Velea et al. [26] report an OpenCL version of ChaCha20 that achieves up to 16 Gbps. However, to the best of our knowledge, no other implementations of ChaCha20 on the GPU are publicly available. Therefore, we adapt D.J. Berstein's publicly available reference implementation of ChaCha20[2] for the GPU using CUDA.

III. GRBC: GPU-Accelerated Response-Based Cryptography

A. Noise Injection

Our algorithm performs all operations in software and does not use a hardware PUF. To simulate the client's key generated by a PUF, we begin by arbitrarily selecting the client's 256-bit key, K_c. Next, we select a Hamming distance, d, and flip d bits in K_c to produce K_s. The flipping of d bits in K_c represents the inherent noise in a hardware PUF. In our experimental evaluation (Section IV), we test the performance of our algorithm with varying levels of noise. For instance, if we flip d=5 bits, then we obtain $5/256 \approx 0.02$ (2%) noise in K_s. In this example, for the server to be guaranteed to recover K_c using K_s, all keys within a Hamming distance of 5 of K_s must be searched.

B. Partitioning and Iteration over the Key Space

To authenticate the client's private key, K_c, the server uses a client-provided user ID, U, PUF-generated challenge key, K_s, and ciphertext, C_c (see Section II-B). This challenge key, K_s, is assumed to be at most a Hamming distance, d, away from the client's private key, K_c. When using a PUF, the maximum value of d would be based on the PUF's error rate. Therefore, the server must iteratively correct the bit errors in this challenge key and encrypt the plain text sent from the client with this corrected key to produce a ciphertext, C_s, that is equal to the client-provided ciphertext, C_c. Since, in practice, the number of error bits may be less than the PUF's error rate, it will be required to iterate all Hamming distances up to and including the maximum Hamming distance, d. Consequently, for each Hamming distance $j \leq d$, GRBC will define a key space, P_j, as a set of bit strings that uniquely differ by j bits from the client's private key. This implies that for each $j \leq d$, the size of the key space is:

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

Then, at each Hamming distance $j \leq d$, the server parallelizes the authentication process by partitioning the corresponding key space, P_j. To process this partitioned key space, GRBC executes a single GPU kernel. This means that each thread, executing as a result of a kernel invocation, is assigned one of the key sets of the partitioned key space. We denote the number of threads to be used for a GPU kernel as t and calculate the size of these key sets (or the number of keys assigned to each thread) as follows:

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

For illustrative purposes, and without the loss of generality, we assume that $|P_j| \bmod t$=0 (in practice, if $|P_j| \bmod t \neq 0$, then we assign these remaining keys to threads to be processed). Two arguments that determine the size of t are specified for every GPU kernel invocation: the number of threads per block and the number of blocks. The number of blocks is calculated by dividing $|P_j|$ by a selected number of keys per block, where a good value of this parameter is platform-specific (we derive good values for t in our evaluation).

To iterate the key space, for every $j \leq d$, we enumerate the set P_j. We then map each index of this enumeration to a unique bit string combination, as is illustrated in Knuth [27], and assign a starting and ending index to each thread executing on the GPU. The assignment of indices for each thread is based on the size of the key sets, n_j, from the partition of P_j. For example, thread 0 will iterate the indices 0 through

[1]https://github.com/zhenlin36/scatter_gather_aes_cuda

[2]https://cr.yp.to/chacha.html

$n_j - 1$, and for each index, thread 0 will decode a unique bit string combination and XOR this combination with K_s to produce an element from the set P_j. Each thread then works on its own set of bit strings in P_j. Threads perform their work without overlapping the work of any other thread by simply knowing the value of n_j and their own thread id. To iterate across its own set of bit strings, a thread uses *Gosper's hack*, as demonstrated by Knuth [28], to compute the next lexicographical combination until it reaches its ending bit string combination.

C. Terminating the Search Early

Given a PUF that uniformly generates error bits (in practice, it would not be uniform), the PUF-generated client key, K_c, will be found at a random location in P_j. Therefore, to minimize the time taken for authentication, the server must terminate the search once a thread finds the correct key. To achieve this, GRBC has the thread that finds the matching cipher update a variable in unified memory, which can be read by all threads executing the RBC search and is accessible across GPU devices. Each thread then checks every r number of iterations whether any other thread has yet to authenticate the client by reading the flag in unified memory. If a thread has successfully authenticated, all other threads return prior to finishing their searches.

D. Fragmentation

The number of keys searched increases exponentially with d (Equation 1). Therefore, only certain classes of computers will be able to perform the search within the threshold time, T=5 s. For example, a laptop computer may be able to compute up to d=3, a workstation up to d=4, and a cluster would be required at d=5, and any Hamming distance d>5 would be intractable for the RBC search on a modern computer. To address this limitation, we use a fragmentation scheme [2] that reduces the search space defined by Equation 1. The main idea is to generate a 256-bit private key on the client device (K_c), which is then fragmented into g subkeys, each of which is 256-bits. The bits that are not generated by the PUF use error-free padding using a scheme only known to the client and server. Then, g keys are sent to the server for authentication, and the RBC search is carried out on the g keys, but the padded values are not searched. The protocol that is used in this work to concurrently select the same cells in the image of the PUF is Ternary Addressable Public Key Infrastructure (TA-PKI), which contains shared information that is used to generate the error-free padding for each authentication cycle [29].

If K_c is the 256-bit key generated by the client's PUF, we let K_i be a subkey generated using fragmentation where i=1, . . . , g, and g is the number of subkeys generated from K_c. We assume that g evenly divides 256. Figure 2 shows K_c fragmented into g=2 subkeys, where the padding is denoted as "1010. . . 10"[3].

[3]This is for illustrative purposes; in practice, the padding would not simply be a series of alternating values.

K_C : 1111111100000000 0000000011111111

F_1 : 1111111100000000 1010101010101010

F_2 : 0000000011111111 1010101010101010

Fig. 2. Example of splitting K_c into g=2 keys using fragmentation. For illustrative purposes, K_c is a 64-bit key instead of a 256-bit key.

Without fragmentation, the average number of keys searched (Equation 1) requires searching half the key space for the key. In contrast, with fragmentation, there may not exist a key in the key space that yields half of the keys searched for a given d and g. Consequently, a better metric to consider when using fragmentation is the upper bound on the number of keys searched. This occurs when all error bits are located within the same subkey. We define this upper bound as follows:

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

where $g \leq \frac{256}{2d}$ for a 256-bit key[4]. Being unaware of the padding scheme, an opponent would need to search g keys of size 256 bits implying a key search complexity of $O(g \cdot n_o)$.

Fragmentation enables tuning the protocol using d and g to have more control over the complexity of the search, which enables the RBC engine to be deployed on a more granular range of compute capabilities.

E. GPU AES Encryption and Kernel Optimizations

We adapt the baseline AES implementation described in Section II-D to perform round-wise key expansion. Partially expanding the key every round uses only 64 bytes of memory compared to expanding the key at once, which requires 240 bytes of memory. Additionally, we optimize our GPU kernel by transferring the substitution table (or S-box) used in AES to shared memory on the GPU.

F. Algorithm Overview

Algorithm 1 outlines GRBC, which accelerates RBC by executing an enumeration scheme for iterating large key spaces, deploying an early exit strategy, optimizing the AES baseline implementation and utilizing multiple GPUs.

Algorithm 1 begins on line 2 by initializing a variable for counting the number of authentications made. Line 3 then loops across each subkey, $K_1, \ldots, K_g$. Then, a subkey, K_i, is generated using K_s, d, and g, which takes a subkey from K_s and applies error-free padding (line 4). Line 5 then initializes the key that will store the client's private key (if found) for authenticating the client, and line 6 initializes the flag for terminating the search early. Then, on line 7, a ciphertext, C_s, is generated by encrypting the client's user ID, U, with the current subkey, K_i. Line 8 checks for equality between C_s and the client's ciphertext for the current fragmentation

[4]We constrain $g \leq \frac{256}{2d}$ for 256-bit keys because if $g > \frac{256}{2d}$, then the upper bound does not hold for those values of d and g.

Algorithm 1 GRBC Algorithm

```
1:  procedure GRBC(U, C_c, K_s, d, g)
2:      authCount ← 0
3:      for i ∈ 1,...,g do
4:          K_i ← genFragKey(K_s, d, g)
5:          authKey ← ∅
6:          keyFound ← False
7:          C_s ← encrypt(U, K_i)
8:          authCount = authCount + (C_s = C_c[i − 1])
9:          if C_s ≠ C_c[i − 1] then
10:             for j ∈ 1,...,d do
11:                 |P_j| ← computeNumKeys(d)
12:                 t ← computeNumThreads(|P_j|)
13:                 n_j ← computeKeysPerThread(|P_j|, t)
14:                 GPURBCKernel(K_i, authKey, keyFound, j, U, C_c[i−
    1], t, n_j)
15:                 authCount = authCount + (authKey ≠ ∅)
16:     return (authCount = g)

17: procedure GPURBCKernel(K_i, authKey, keyFound, j, U, C_c, t, n_j)
18:     tid ← getThreadId()
19:     C_s ← ∅
20:     (startComb, endComb) ← getCombPair(tid, j, n_j)
21:     (currKey, endKey) ← initKeyIterator(K_i, startComb, endComb)
22:     while !(earlyExit and keyFound) and (currKey ≠ endKey) do
23:         C_s ← encrypt(U, currKey)
24:         if C_c = C_s then
25:             authKey ← currKey
26:             if earlyExit then
27:                 keyFound = True
28:         currKey ← getNextKey()
29:     return
```

iteration, $C_c[i-1]$, and the result is added to authCount. If C_s is equivalent to $C_c[i-1]$, then the search is terminated (line 9). Otherwise, the key search for authenticating the client begins on line 10. This leads to looping across an increasing Hamming distance, j, until reaching the limit, d[5]. Then several values for partitioning and assignment of work to threads are calculated on lines 11, 12, and 13, which refer to the number of keys to process, the total number of threads to launch in the kernel, and the number of keys computed per thread. The latter two values are used as input to the GPU RBC kernel (line 14). Line 15 then checks if the resulting authentication key, authKey, is empty, and the result of this comparison is added to authCount. Finally, on line 16, authCount is compared to g, and this result is returned.

We outline GPURBCKernel as follows (line 17). The GPU kernel calculates each thread's id on line 18, and the resulting cipher from encrypting the client's user ID, U, is initialized on line 19. Then, on line 20 the starting and ending bit string combinations are calculated by using the thread's id, tid, the current Hamming distance, j, and the number of keys per thread, n_j. Any given thread processes its own key search space by iterating from a starting key to an ending key. Hence, an iterator is created on line 21. Until the current key is equal to the ending key, a thread loops across its key search space as shown on line 22. Upon each iteration, a thread encrypts the client's user ID, U, with the current key, currKey, to produce a cipher, C_s. This cipher is compared with C_c for equality on line 24. If there is a match, then the authentication key, authkey, is set to the current key, currKey. Then, if a global value, earlyExit, is set to true, line 27 sets the flag keyFound to true to end the execution of all running threads. If there was either no match between the ciphers or earlyExit was set to false, then on line 28 the next key in the thread's search space is acquired. If a match never occurs for the given thread, then authKey remains unchanged, and the kernel exits.

To accommodate multiple GPUs, Algorithm 1 is reconfigured as follows. An authentication key, authKey, is initialized for each GPU device on line 5. Next, lines 11 and 13 are partitioned and distributed to a GPU kernel invocation (line 13) for each device. Finally, line 15 is changed by adding to authCount a union of equivalence comparisons between the authKey and K_c for each GPU.

IV. Experimental Evaluation

A. Experimental Methodology

We employ five platforms that are detailed in Table I. Plat1 is equipped with a single Pascal generation GPU (GP100), Plat2 is equipped with two Pascal TitanX GPUs, Plat3 is equipped with four Volta GPUs (V100), and Plat4 and Plat5 each have 28 Intel Xeon CPU cores of the Skylake and Broadwell architectures, respectively. We use Plat3 to assess the scalability of our algorithm as executed on multiple GPUs within a single computer, and Plat4 and Plat5 to examine the performance of the MPI implementation. Plat3, Plat4, and Plat5 are part of our institution's cluster that is shared by multiple users, and we ensure that 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.

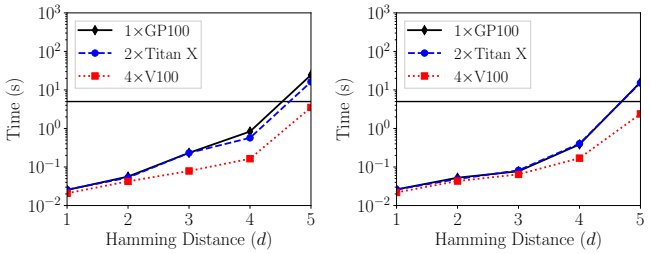
GRBC: Our GPU-accelerated RBC algorithm. Unless otherwise noted, we present our results using AES, as results are similar for ChaCha20 and SPECK. Recall from Section III-B that n_j is the number of keys each GPU thread searches. In all experiments using AES, we set n_j=$8,192$ on Plat1 and Plat2, and set n_j=$1,024$ on Plat3. In all experiments using ChaCha20, we set n_j=$1,024$ on Plat1 and Plat2, and set n_j=$1,024$ on Plat3. In all experiments using SPECK, we set n_j=$1,024$ on Plat1, set n_j=$2,048$ on Plat2, and set n_j=512 on Plat3. These values are derived in Section IV-B4. Also, r is the number of iterations before checking if any other thread has found the client's private key (Section III-C). In all experiments with an early exit procedure, we set r=1 as it has the highest frequency of checks and we observed a low variation between response times for varying values of r.

CPU-MPI: Our distributed-memory reference implementation that uses OpenSSL. The algorithm uses MPI for parallelizing the search on multiple computers. For experimental

[5]In practice, d and g will be set to a Hamming distance and fragmentation level that correspond to an achievable authentication timing threshold for a given platform.

TABLE I
DETAILS OF THE PLATFORMS USED IN THE EXPERIMENTAL EVALUATION. ALL PLATFORMS USE INTEL XEON PROCESSORS.

	CPU					GPU			
Platform	Model	# Nodes	Cores (Total)	Clock	Memory	Model	Cores/GPU	Memory/GPU	CUDA
PLAT1	2×E5-2620 v4 (Broadwell)	1	2×8 (16)	2.1 GHz	128 GiB	1×Quadro GP100	3584	16 GiB	v.9
PLAT2	2×E5-2683 v4 (Broadwell)	1	2×16 (32)	2.1 GHz	256 GiB	2×Titan X	3584	12 GiB	v.10.1
PLAT3	2×Gold 6132 (Skylake)	1	2×14 (28)	2.6 GHz	384 GiB	4×Tesla V100	5120	16 GiB	v.11.2
PLAT4	2×Gold 6132 (Skylake)	19	2×14 (532)	2.6 GHz	196 GiB	-	-	-	-
PLAT5	2×E5-2680 v4 (Broadwell)	1	2×14 (28)	2.4 GHz	128 GiB	-	-	-	-



(a) Exhaustive search (b) Average case search

Fig. 3. Response time vs. Hamming distance d for GRBC on each of our GPU-equipped platforms, comparing the (a) exhaustive search; and (b) average case search. We do not use fragmentation (g=1) for all platforms. Experiments performed using the AES block cipher.

purposes, the algorithm only uses AES. Due to space constraints we are unable to show optimizations to the algorithm. However, we optimize the early exit search procedure to nearly eliminate the overhead of communicating that the key was found.

B. Evaluation of GRBC

We evaluate GRBC and demonstrate searches with the following features: (i) exhaustive searches that search all keys in the key space up to the maximum Hamming distance, d; (ii) key searches where the client's key is found in the middle of the key space, thus representing the average case search (Equation 1); and (iii), the case where the key is randomly found within the key space. Additionally, we compare key throughput as the number of keys computed per second, and we discuss whether various configurations are able to authenticate a user within T=5 s. Comparing the above characteristics allows us to understand how the algorithm will perform when deploying the RBC engine in practice.

1) Comparison of Exhaustive and Average Case Performance: In this section, we compare the exhaustive and average case searches, where the former computes all keys in the key space, and the latter terminates the search when the correct client key is found. Also, we do not use fragmentation and set g=1. The only parameter needed to tune the performance of GRBC is n_j, which is the number of keys assigned to each thread. We use the values of n_j outlined in Section IV-A that will be derived in Section IV-B4.

Figure 3(a) plots the response time vs. the Hamming distance, d, for the exhaustive search, where the horizontal line demarcates the T=5 s authentication threshold. We find that for an exhaustive search, all GPU-equipped platforms can authenticate the user within $T \leq 5$ s when $d \leq 4$. However, only with the 4×V100 GPU platform can we authenticate a user within $T \leq 5$ s when d=5.

Figure 3(b) shows the same plot as Figure 3(a), but for the average case search where the key is found in the middle of the key space, thus terminating the search early. We observe the same phenomena as in Figure 3(a), where only the 4×V100 GPU platform can authenticate a user within T=5 s when d=5, and that all platforms are capable of authenticating within time constraints when $d \leq 4$.

This demonstrates that for T=5 s, only one GPU platform is guaranteed to be able to authenticate a user within time constraints for both the average and exhaustive search cases at d=5. This also illustrates that not using fragmentation (g=1) is a restrictive configuration for RBC in practice, as the key search space increases exponentially, which makes d>5 intractable for the platforms used in this evaluation when T=5. Therefore, these results motivate the use of fragmentation, which we explore in the next section.

2) Employing Fragmentation: Figure 4 plots the response time vs. Hamming distance on all GPU-equipped platforms for selected fragmentation levels, g, where the average case number of keys, $n_s(d)$, is searched without fragmentation, (g=1, Equation 1), and the upper bound number of keys, $n_s(d, g)$, is searched when using fragmentation (g>1, Equation 5). We observe in Figure 4 that increasing g enables more control over the complexity of the RBC search. In particular, across all platforms, we find that when g=8, $d \leq 13$ is tractable within T=5 s, and when g=4 that $d \leq 7$ is tractable.

Using a T=5 s authentication threshold, without fragmentation, the Hamming distance is constrained to $d \leq 5$. With fragmentation, there is a larger parameter space of (d, g) pairs such that the RBC protocol can be deployed on a larger range of computer systems.

3) Random Key Generation: The exhaustive and average cases provide performance bounds on GRBC but do not capture the full range of expected response times (e.g., when the corrupted bits are in the beginning portion of the key). Figure 5 plots the response time vs. d, showing different levels of fragmentation ($g \in \{1, 2, 4, 8\}$) where the search is performed on a key with d randomly-flipped bits. For each (d, g) pair, the response time is shown for 75 randomly generated keys. Across all values of g, we find that as the key space increases (increasing d), there is a larger range of response times. Also, observe that the points cluster around different time ranges. This reflects that the response time is sensitive to the number of error bits in each subkey. In one extreme, if all error bits fall within a single subkey, then the response

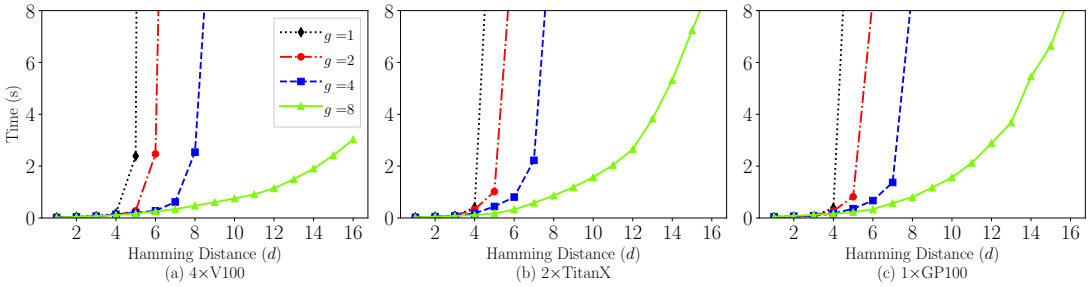


Fig. 4. Response time vs. Hamming distance for selected fragmentation levels, g. For g=1 the average case is conducted using Equation 1, and for all g>1, an exhaustive search is carried out using our upper bound from Equation 5. Experiments performed using the AES block cipher.

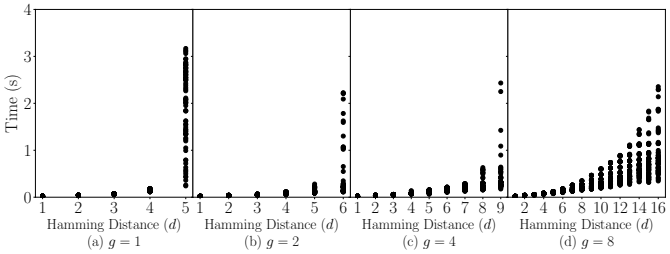


Fig. 5. Spread in response times vs. Hamming distance (d) showing different levels of fragmentation where the search is performed on a key with d randomly-flipped bits. We show results using PLAT3. Values of d with $T \leq 5$ s are shown. Experiments performed using the AES block cipher.

time is high, whereas at the other extreme, if they are evenly distributed into each subkey, then the response time is low. Intermediate distributions of errors in subkeys are shown by response times between these two extremes.

4) Assigning Keys to Threads: The selection of a good value of n_j is platform-dependent. Figure 6 plots the key search throughput in keys/s vs. the number of keys assigned to each thread (n_j) on all GPU-equipped platforms for an exhaustive search with g=1 and d=5. We observe that a low value of n_j increases the total number of threads, which increases thread management overhead. If we assign too many keys per thread, then there may be too few threads to saturate GPU resources. Note that we do not examine $d \geq 6$, as it was shown to be intractable on any of the GPU-equipped platforms within T=5 s (Figure 3). If we let j=1, . . . , d refer to each Hamming distance searched up to a maximum Hamming distance d, the greatest search time occurs when j=d. Therefore, the parameter n_j should be tuned to achieve the best performance when j=d, and not j<d. Consequently, in all of our experiments, we used a good value of n_j when d=5 as derived from Figure 6. Particularly, the GP100 and Titan X have the highest throughput when n_j=2^{13}. For the V100, we find n_j=2^{10} yields the best performance.

C. Comparison of GRBC to CPU-MPI

We compare GRBC to CPU-MPI where we perform an exhaustive search up to d=5. Figure 7(a) plots the throughput in keys searched per second for all platforms with a GPU compared to CPU-MPI for p=32, 128, 256, 512 ranks.

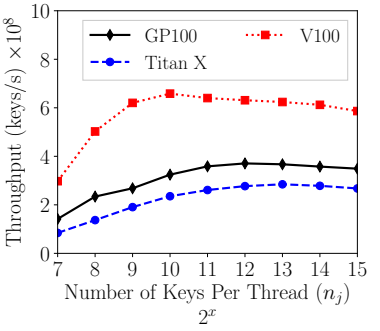


Fig. 6. Throughput in keys searched per second vs. the number of keys each thread searches (n_j), comparing the different GPUs found on our GPU-equipped platforms. For each GPU architecture, we show results for an exhaustive search with d=5, and g=1. Experiments performed using the AES block cipher.

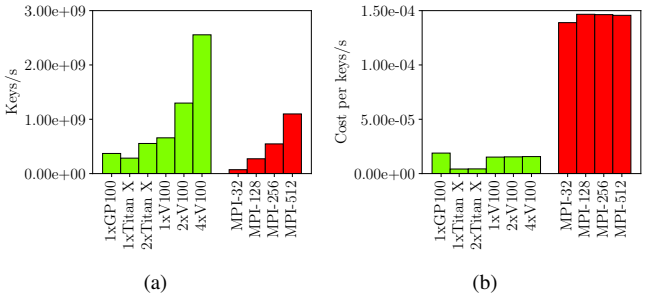


Fig. 7. (a) Throughput in keys searched per second of GRBC and CPU-MPI for an exhaustive search using a Hamming distance d=5. (b) Cost per key/second in dollars USD of GRBC and CPU-MPI in (a). GRBC (CPU-MPI) is executed on PLAT1, PLAT2, and PLAT3 (PLAT4). Experiments performed using the AES block cipher.

We find that GRBC significantly outperforms CPU-MPI. Comparing a single GP100 and V100 GPU to CPU-MPI with p=32 (roughly a single compute node), GRBC achieves 5.16× and 9.17× the throughput, respectively, as CPU-MPI. Using the data from Figure 7, we find that ∼300 CPU cores can be replaced by a single V100 GPU. Figure 7(b) plots the approximate cost per key searched per second comparing GRBC to CPU-MPI. The GPU is significantly more cost-effective, particularly the consumer-grade TitanX GPU.

D. Contrasting GRBC using AES, SPECK, and ChaCha20

To demonstrate the generalizability of our GPU-accelerated algorithm, we compare GRBC using AES, SPECK, and ChaCha20. Figure 8 plots the throughput in keys searched per second for all symmetric ciphers across all platforms with a GPU where we perform an exhaustive search up to d=5. We find that the SPECK cipher outperforms AES and ChaCha20

on all platforms and that ChaCha20 outperforms AES on both PLAT2 and PLAT3. While SPECK is a lightweight cipher offering high throughput for GRBC, AES offers higher levels of security at the cost of slightly lower throughput.

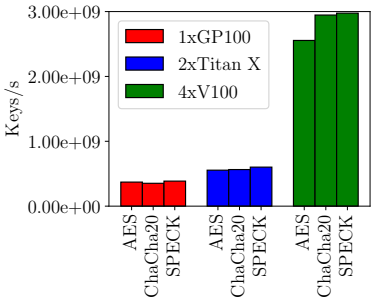


Fig. 8. Throughput in keys searched per second for an exhaustive search with a Hamming distance d=5 using AES, ChaCha20, and SPECK. GRBC is executed on PLAT1, PLAT2, and PLAT3 corresponding to the left, middle, and, right bar groups, respectively.

V. Discussion and Conclusions

We presented GRBC, the first paper proposing to use GPGPU for RBC. The engine enables a client device equipped with a PUF to authenticate and securely communicate with a server. Since the client's private key differs each time it is generated by the PUF, a search must be employed on the server to find the key that matches the client's initially recorded challenge. We compared the performance of GRBC on three platforms and found that all GPUs yield high key search throughput for 3 different symmetric cipher encryption algorithms. We find that we can authenticate a user well within reasonable time constraints.

We find that GRBC is more cost-effective than previous RBC implementations, where one NVIDIA V100 GPU demonstrates equivalent search throughput to approximately 300 CPUs, enabling practical single-node RBC implementations.

Acknowledgment

We thank Frédéric Loulergue for the use of his machine and Chris Coffey for cluster support.

References

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

[2] B. Cambou, "Unequally powered cryptography with physical unclonable functions for networks of internet of things terminals," in *2019 Spring Simulation Conf.*, 2019, pp. 1–13.

[3] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld, "Physical one-way functions," *Science*, vol. 297, no. 5589, pp. 2026–2030, 2002.

[4] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas, "Silicon physical random functions," in *Proc. of the 9th ACM conference on Computer and communications security*. ACM, 2002, pp. 148–160.

[5] S. Chen, B. Li, and Y. Cao, "Intrinsic physical unclonable function (PUF) sensors in commodity devices," *Sensors*, vol. 19, no. 11, p. 2428, 2019.

[6] M. Hofer and C. Böhm, "Error correction coding for physical unclonable functions," in *Austrochip, Workshop on Microelectronics*, 2010.

[7] J. Delvaux, D. Gu, D. Schellekens, and I. Verbauwhede, "Helper data algorithms for PUF-based key generation: Overview and analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 6, pp. 889–902, 2014.

[8] G. T. Becker, A. Wild, and T. Güneysu, "Security analysis of index-based syndrome coding for PUF-based key generation," in *2015 IEEE Intl. Symposium on Hardware Oriented Security and Trust*. IEEE, 2015, pp. 20–25.

[9] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck, "Cryptographics: Secret key cryptography using graphics cards," in *Cryptographers' Track at the RSA Conf.* Springer, 2005, pp. 334–350.

[10] A. D. Biagio, A. Barenghi, G. Agosta, and G. Pelosi, "Design of a parallel AES for graphics hardware using the CUDA framework," in *2009 IEEE Intl. Parallel Distributed Processing Symposium*, 2009, pp. 1–8.

[11] D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright, "Fast software AES encryption," in *Intl. Workshop on Fast Software Encryption*. Springer, 2010, pp. 75–93.

[12] K. Iwai, T. Kurokawa, and N. Nisikawa, "AES encryption implementation on CUDA GPU and its analysis," in *2010 First Intl. Conf. on Networking and Computing*. IEEE, 2010, pp. 209–214.

[13] C. Mei, H. Jiang, and J. Jenness, "CUDA-based AES parallelization with fine-tuned GPU memory utilization," in *2010 IEEE Intl. Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2010, pp. 1–7.

[14] J. Gilger, J. Barnickel, and U. Meyer, "GPU-acceleration of block ciphers in the OpenSSL cryptographic library," in *Intl. Conf. on Information Security*. Springer, 2012, pp. 338–353.

[15] A. Khan, M. A. Al-Mouhamed, A. Almousa, A. Fatayar, A. Ibrahim, and A. Siddiqui, "AES-128 ECB encryption on GPUs and effects of input plaintext patterns on performance," in *15th IEEE/ACIS Intl. Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. IEEE, 2014, pp. 1–6.

[16] W.-K. Lee, H.-S. Cheong, R. C.-W. Phan, and B.-M. Goi, "Fast implementation of block ciphers and PRNGs in Maxwell GPU architecture," *Cluster Computing*, vol. 19, no. 1, pp. 335–347, 2016.

[17] A. A. Abdelrahman, M. M. Fouad, H. Dahshan, and A. M. Mousa, "High performance CUDA AES implementation: A quantitative performance analysis approach," in *2017 Computing Conf.*, 2017, pp. 1077–1085.

[18] A. A. Abdelrahman, H. Dahshan, and G. I. Salama, "Enhancing the actual throughput of the AES algorithm on the Pascal GPU architecture," in *2018 3rd Intl. Conf. on System Reliability and Safety*, 2018, pp. 97–103.

[19] Nvidia, "CUDA Toolkit Documentation: Performance Guidelines," 2019, accessed: May 24, 2020. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[20] A. Munshi, "The OpenCL specification," in *2009 IEEE Hot Chips 21 Symposium*. IEEE, 2009, pp. 1–314.

[21] OpenSSL, "OpenSSL Webpage," accessed: May 24, 2020. [Online]. Available: https://www.openssl.org/

[22] Z. Lin, U. Mathur, and H. Zhou, "Scatter-and-Gather Revisited: High-performance side-channel-resistant AES on GPUs," in *Proc. of the 12th Workshop on General Purpose Processing Using GPUs*. ACM, 2019, pp. 2–11.

[23] Z. H. Jiang, Y. Fei, and D. Kaeli, "A complete key recovery timing attack on a GPU," in *2016 IEEE Intl. symposium on high performance computer architecture*. IEEE, 2016, pp. 394–405.

[24] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK Families of Lightweight Block Ciphers," Cryptology ePrint Archive, Report 2013/404, 2013, https://eprint.iacr.org/2013/404.

[25] D. J. Berstein, "Chacha, a variant of salsa20," in *Workship Record of SASC*, 2008.

[26] R. Velea, F. Gurzau, L. Margarit, I. Bica, and V. Patriciu, "Performance of parallel chacha20 stream cipher," *2016 IEEE 11th Intl. Symposium on Applied Computational Intelligence and Informatics*, pp. 391–396, 2016.

[27] D. E. Knuth, *Generating all combinations and partitions*. Addison Wesley Professional, 2010.

[28] ——, *The Art of Computer Programming*. Addison Wesley Professional, 2009, vol. 4.

[29] B. Cambou and D. Telesca, "Ternary computing to strengthen information assurance. development of ternary state based public key exchange," in *IEEE, SAI-2018, Computing Conf.*, 2018.