A Response-Based Cryptography Engine in Distributed-Memory

Christopher Philabaum, Christopher Coffey, Bertrand Cambou, and Michael Gowanlock

School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, U.S.A.
{cp723, Chris.Coffey, Bertrand.Cambou, Michael.Gowanlock}@nau.edu

Abstract. Cryptographic keys extracted from Physical Unclonable Functions (PUFs) can be produced reliably when paired with helper functions, but this places a burden of computation on client devices. With the disparity in power between weaker Internet of Things devices and the more powerful server clusters, response-based cryptography (RBC) shifts that burden of error correction on the server to match the client's response. Noise injection is a potential solution for security in hostile environments, so it is vital to know to what error rate can a distributed system correct in an RBC cyber system. In this paper, we explore the feasibility and scalability of response-based cryptography in a high-performance computing environment. We present a highly parallel, MPI-based implementation using up to 512 ranks/cores. Scalability was achieved by ordering the key space lexicographically and having each rank independently generate its own work using combinadics, where we assign equal workloads to each MPI rank. Terminating the key search early across distributed-memory ranks is challenging as it can incur significant overhead. Thus, we compare two strategies for terminating the search algorithm early. We assume that a typical user prefers a service to be responsive within a two second window. We are able to achieve authentication under this assumed latency metric up to 5 bit errors over an AES-256 key when utilizing 512 ranks. The speedup our RBC search algorithm developed achieves good scalability yielding a speedup of 404× on 512 ranks.

Keywords: Cryptographic protocols, Cryptography, Distributed computing, High performance computing, Response-based cryptography

1 Introduction

Physical Unclonable Functions (PUFs) have repeatedly been shown to provide strong, secure sources of authentication [18, 22, 31]. Various practical implementations have been proposed, such as flash [33], SRAM [19,25,36], DRAM [27,37], MRAM [39], ReRAM [11,15], ring oscillators [34], and Monte Carlo [2,35]. However on their own, PUFs are more often than *not* absolutely reliable as attributed

to their physical nature. By contrast, a protocol that uses PUFs to derive cryptographic keys must be pristine. If at least any single-bit error occurs, the design of a strong cipher *should* produce an avalanche effect over the resulting ciphertext's bitstream, i.e., the ciphertext will change drastically from any minimal change to the key. There are other approaches that exist that use helper functions and error correction codes [5, 6, 16, 17, 23, 26, 30, 38, 40] sent from server to client-side. It is then the client's job to correct for its own noisy key and match the server's produced key.

Placing the burden on the client exhibits several design flaws. The client device is more naturally exposed to side-channel attacks (or other such related cryptanalytical attacks) just by the nature of typically being mobile. The helper function yields costs in both computational resources, which requires more power consumption within a more constrained environment

Response-based cryptography [9, 12–14] flips the cost of computation to the server-side. This also opens up the possibility for purposeful noise injection techniques when the client device is exposed to more hostile environments. When information must be critically secure in such conditions, one can design a system with more computational resources to secure the connection. Thus without access to the PUF and its secret parameters or a similar amount of computing power, an attacker would be unable to break such a system through direct means.

To our knowledge, no implementation of a response-based cryptography engine has targeted a distributed-memory system, nor has any experimental analysis been done on the problem of scalability. The motivation is then to investigate how much error rate can a typical high-performance computing cluster (HPC) handle within in a "reasonable" time frame. We define an authentication time window of $T = 2$ seconds, based on real world thresholds for services toward end users similarly target this time frame. This paper makes the following contributions:

- We motivate and propose a key iteration strategy that allows us to independently iterate over the key space, such that each process has the same workload. To the best of our knowledge, our use of combinadics in the context of high-performance computing is a novel methodology to evenly distribute the workload and have each process independently iterate over all possible cryptographic keys for any given tractable error rate.
- To ensure that we do not waste computational resources after the key has been found by a worker, we propose a communication strategy to efficiently and gracefully end the search between MPI ranks.
- We compare two early exit strategies and achieve an average 78.97% of the peak throughput (keys/second) using our best early exit strategy.
- Our RBC system achieves a speedup of up to 404× on 512 cores. This high level of scalability successfully yields authentication within the $T = 2$ s threshold at 1.65 s.

The paper is organized as follows. In Section 2, we discuss the terminology used and what response-based cryptography is and how it works. In Section 3,

we explain the methodology chosen for both the key iteration and the early exit strategy for aborting the search when the key is found. In Section 4, we describe the implementation and how the benchmarks are designed. With Section 5, first we explain the compute cluster used and its hardware specifications for our experiments. We follow by showing the results of our experiments, evaluating the impact of the number of iterations between communications in Section 5.3, comparing the viability of two early exit strategies in Section 5.4, and analyze the latency, speedup, and parallel efficiency for up to 512 cores at a Hamming distance of 5 in Section 5.5. Finally, we conclude and summarize our findings and discuss possible future work in Section 6.

2 Background

2.1 Terminology

To avoid confusion, the following terminology will used throughout the remaining text:

- "***Eve***" refers to a malicious entity or attacker whose goal is to break the security of the client device and the server.
- The cryptographic key "K_A" is defined as the key that the client produces from an unreliable source such as a PUF. We denote this as the "***response***".
- The cryptographic key "K_B" is defined as the key that the server produces from the enrollment of the client device. This can also be thought of as the "***challenge***".
- "C_A" and "C_B" represent the client's and server's ciphertexts, respectively.
- The "***enrollment***" is defined as a snapshot or a series of snapshots of the client's PUF. This is the data stream that the server generates the cryptographic key K_B from.
- "***Response-based cryptography (RBC)***" refers to the concept of the server performing the *correction on the response*, rather having the client device performing the correction with a helper function or other methodology.
- The "***RBC engine***" the core algorithm and focus of this paper, whose job is to search for K_A, derived from the server's own K_B.

2.2 Response-Based Cryptography

The focus of response-based cryptography is to flip the asymmetric load for key error correction to be performed on the server as opposed to the client. This means that the client's key is static, whereas the server must perform the work to iteratively search for the potentially slightly corrupted key. As seen in Figure 1 (Top), the server starts by assuming the client's key is a pristine copy that exactly matches the server's as shown by the "starting point". This first key K' is used to create a new ciphertext using the cipher E (such as AES-256). $C = E_{K'}(P)$ thus represents the ciphertext C that is created by the cipher E,

the key K', and plaintext P. In the context of a symmetric cipher, the plaintext P is chosen to be the client's identification number that is also known by the server. If a match between ciphertexts is found, then the server authenticates the client, and both devices may use this key as a means to secure their messages.

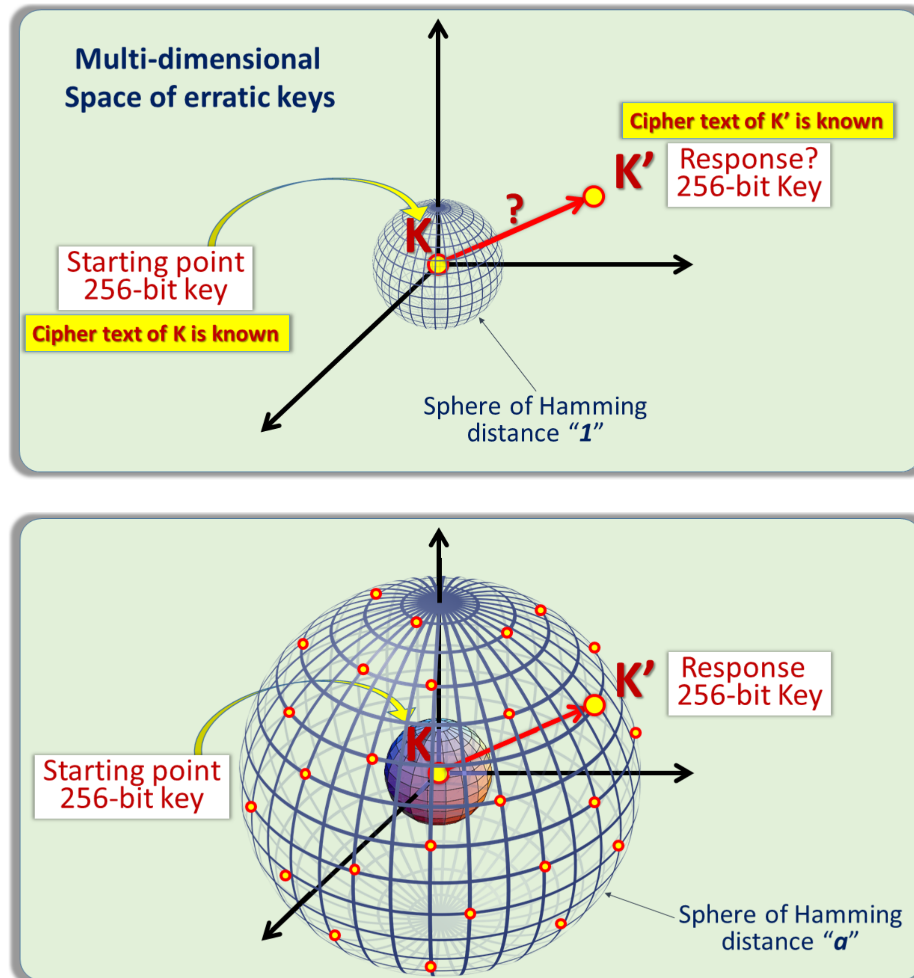

Fig. 1: A visual representation of the key search space by iterating through every combination of bit errors with Hamming distance D relative to the starting key. (Top) The early step of the algorithm where we only consider all keys K' with a Hamming distance of 1 relative to K. (Bottom) A further expansion of the key search space up to an arbitrary Hamming distance a. This demonstrates how this search space widens with each increase in Hamming distance.

However, the inherent error-prone nature of PUFs typically creates a mismatch between the client's key K_A and the server's key K_B. The server thus must perform an iterative search through all possible corruptions of the client's key to find a suitable match. In Figure 1 (Bottom), the Hamming distance is increased to $D = 1$; every corruption with this many bit errors is tested. If no matching keys are found, then the Hamming distance is in turn increased by $D + 1$. This process is continued up to a matching key found, $D > 256$, or the time threshold, T, runs out. If this cannot be done in a reasonable time, in this case arbitrarily defined to be $T = 2$ seconds, then the client is not authenticated and must try again. At the same time, the attacker Eve wants to break the cyber system. While E, C_A, and P are assumed to be public information, the keys K_A and K_B should be private relative to Eve. Even if Eve suspects a reasonable Hamming distance of less than 5 errors, she has no relative starting position in the key space.

On the other hand, the server has a "head start". Unlike Eve who must choose some starting point in the key search space at random, the server knows what the client key K_A must resemble. Let us say the server's key is $K_B = 10011010_b$. In the worst case scenario, the client's device is 100% noisy and can equally produce all combinations of errors. Then the server must search through a Hamming distance of $0, 1, \ldots, 8$ until they find that $K_A = 01100101_b$; note that the search space size is equal to that of Eve. This translates to Equation 1 on average for both the server and Eve. Since the error rate is absolute, the all possible combinations must be considered equally probably. This is equivalent to enumerating to entire number line of all possible combinations of errors, where the average case is half way along this number line. In other words, if all information about the error properties of the PUF is kept secret, the attacker must search through all combinations of Hamming distance 0 to 8. Their average runtime is the half of all these combinations, given that as far as they are concerned, any number of errors can occur.

Now consider the average case scenario for an SRAM PUF-originated key that on average produces a bit error rate of 25% (we use an unrealistically noisy source for illustrative purposes). Now instead the server has to explore only up to and including a Hamming distance of 2; the server's average key search space is defined by Equation 2.

Note that in reality the bit error rate is not fixed. The true average will fluctuate based on the distribution of errors for every cell of a PUF. Furthermore, if the exact errors e were known ahead of time, then only $\binom{8}{e}/2$ keys would have to be searched on average.

$$\sum_{i=0}^{8} \binom{8}{i} / 2 = 2^{8-1} = 128 \text{ keys} \tag{1}$$

$$\sum_{i=0}^{1} \binom{8}{i} + \binom{8}{2} / 2 = 1 + 8 + \frac{28}{2} = 23 \text{ keys} \tag{2}$$

Already for *8-bit* keys there is a large disparity of work between the server and Eve, specifically 18% of Eve's key search space is necessary to search. When the same idea is applied to 256-bit cryptographic keys with a much more realistic and tame bit error rate for an SRAM PUF source of 3.69% (based on averaging Schrijen's and Leest's average results at room temperature [36]), this gap grows to Equation 3 for Eve vs. Equation 4 for the server. Clearly one is much more feasible than the other. When combined with a ternary PUF protocol, the bit error rate can be dropped to single digit errors and thus make the problem for the server tractable and Eve still incomprehensibly intractable [3,4,7,8,10,24,32].

$$\sum_{i=0}^{256} \binom{256}{i} / 2 = 2^{256-1} \approx 5.79 \times 10^{76} \text{ keys} \tag{3}$$

$$\sum_{i=0}^{10} \binom{256}{i} + \binom{256}{11} / 2 \approx 3.41 \times 10^{18} \text{ keys} \tag{4}$$

3 A Distributed-Memory Response-Based Cryptography Engine

In the purest and simplest sense, a server that authenticates in an RBC cyber system would iterate through the key space from the starting point in a sequential manner. However, even though the server's search space is far less than that of Eve's at roughly 3.41×10^{18} keys based on Equation 4, this far exceeds the computing capacity of a modern single core processor and exceeds the capacity of modern multi-core computers. Therefore, our solution is to distribute the work of testing every single one of these keys amongst many distributed memory compute nodes as evenly and with as little overhead as possible.

As a general overview, we examine the total key space for Hamming distance D, then splits the work in a balanced way amongst p processes (subsequently referred to as ranks). Each process then independently iterates its own key subspace. To ensure that computation resources are not wasted, if one of the keys manages to find a match, then it is communicated to all other $p - 1$ processes to stop the search early. If none of the processes finds any matching key for Hamming distance p, the distance is increased by 1 and each process generates its own new workload.

Therefore, there are two core mechanics to the algorithm. First, the key iteration step, which includes deriving the next key in the current iteration, encrypting the plaintext with the new key, and comparing this new ciphertext to the client's. Second, each process must communicate to abort the search when commanded by one of the other ranks.

3.1 Key Iteration

The core operation and first half of the algorithm is to first attempt finding the cryptographic key that is originally derived from the server's key, K_B. Each

process is responsible for computing some portion of the key space, ideally with as minimal overlap as possible, by iteratively "corrupting" K_B. Only one such corruption can reproduce the the client's K_A such that $C_A = E_{K_A}(P)$. The approach used in this paper to iterate over every possible key corruption separated the corruption as its own bitstream. For instance, if the Hamming distance between K_A and K_B is D where each key is 256 bits long, then the corruption can be represented by every possible combination of 256-bit bitstrings with D out of all 256-bits set to a value of 1. If the current combination is S_i, the algorithm performs an authentication check by producing the key $K_B \oplus S = K_i$ (for the bitwise XOR operation $\oplus$). The algorithm then tests the question $C_i = E_{K_i}(P) \stackrel{?}{=} E_{K_A}(P) = C_A$, i.e., encrypting the same plaintext P and testing to see if the current key's ciphertext matches the client's ciphertext.

Other approaches considered suffer from the inability to readily enumerate and iterate the key space in a divisible manner. Consequently, we use an approach that exploits the *combinatorial number system*. D. H. Lehmer [21] showed that every combination S corresponds to a unique i; put in the reverse way, for any integer $i \in [0, \binom{\text{Size of bitstream}}{\text{Number of bits set}}]$, we can derive all the bits set in S_i. As illustrated in Knuth [29], there is an algorithm known as `unranking` that for a given non-negative $i < \binom{256}{D}$, we can generate the combination S_i with D bits set in constant time (generalized to $O(n)$ for n-bits). Therefore, a key iteration can be devised by iterating over every integer $i \in [0, \binom{256}{D})$ and deriving the S_i from i. More importantly, every rank knows its own work using only its rank id and Hamming distance D. This results in an embarrassingly parallel search. Each MPI rank can generate its own range of keys using its rank identification number and the Hamming distance D. No communication is required between ranks to generate their work or examining the entire key search space. Instead, the only communication required occurs when a rank finds the key and communicates this information to all other ranks.

Even though the `unranking` algorithm is constant time, there is still a faster method of actively iterating over every combination. With *Gosper's hack* as demonstrated by Knuth [28], only 10 bitwise operations are necessary to take the current combination and compute the next lexicographical combination. Another optimization [1] can add further performance, that uses the `CTZ` (count trailing zeroes) instruction found on modern instruction sets, such as `BMI1` on `x86`.

The final key iteration algorithm is outlined as follows:

1. Have each rank find its starting and ending indices based on $i_0 = \frac{R \cdot \binom{256}{D}}{N}$ and $i_N = \frac{(R+1) \cdot \binom{256}{D}}{N}$.
2. Have each rank find its starting and ending combinations using the `unrank` algorithm on $i_0 \to S_0, i_N \to S_N$.
3. Start at combination $S_{i=0}$.
4. For the current combination, compute the corrupted key with $K_i = K_B \oplus S_i$.
5. Encrypt P so that $C_i = E_{K_i}(P)$.
6. Test if $C_i \stackrel{?}{=} C_A$:

   (a) If the equality evaluates as true, then pass authenticate and signal all other ranks to stop and break.
   (b) If the equality evaluates as false, use Gosper's hack on the current combination S_i to derive S_{i+1} and increment i.
7. If $i \geq i_N$ or the signal is set, break. Otherwise, loop back to Step 4.

3.2 Optimized AES Encryption and 256-bit Integers for RBC

In order to have loop over the appropriate combinations using the methodology described in Section 3.1, the combinations themselves must be interpreted or treated as some integral primitive to perform arithmetic over them. As of the time of this writing, no widely available, general purpose CPU supports instructions such as `CTZ` (count trailing zeros), even for GCC's built-int `int128` data type. This meant that an arbitrary-precision library that supported integral types is necessary, and we describe it as follows.

GMP is a widely supported, stable library that supported all the needs of our implementation. Some limitations, such as a lack of support for `CTZ`, meant replacing the `CTZ`-based implementation with a more inefficient implementation using division instead. However, it was quickly found using profiling that the GMP library spent the majority of its time allocating and deallocating memory, even when reusing the same variables carefully.

While GCC's `int128` was under consideration, it lacked an equivalent `CTZ` instruction. To solve the above issues, we employ a custom fixed 256-bit unsigned integer implementation. To accomplish this, each 256-bit unsigned integer is defined as an array of 4× 64-bit unsigned integers. With static declaration to avoid unnecessary memory allocations/deallocations, our algorithm uses custom versions of bitwise `AND`, `OR`, and `XOR`, `COM` (complement), shift right, shift left, two's complement negation, and comparison operations. We implement `CTZ` by applying the instruction `CTZ` to each 64-bit limb. Addition recovered a low-level add-carry instruction call, that required the carry flag to carry from one limb to the next where necessary. Some portions of GMP remained, allowing for a much simpler implementation of performing a binomial operation that is only done to compute the range of the keys for each rank and not within the main loop itself.

The AES OpenSSL implementation has several performance drawbacks. While functional, profiling also revealed most compute time being spent between allocations and deallocations, even though it uses AES-NI directly. Unsurprisingly, most AES implementations are built with the notion of encrypting/decrypting many, many bytes of data with few keys, not encrypting/decrypting some constant data with many, many keys. To remove OpenSSL out of the the algorithm, Intel's white papers on AES-NI [20] were used as a basis for a very direct, no dynamic memory implementation of preparing the key expansion over the key rounds for the encryption step. The cipher output C_i is then used as the comparison with the client's cipher C_A.

Using the two proposed optimizations, the amount of memory allocations and deallocations done within the iterations themselves is effectively brought

down to zero, excluding any that MPI may perform itself. All the compute time is then focused on readily deriving the next combination from a previous one, applying it to the original server key via `XOR`, and encrypts the shared plaintext P using the new, corrupted key. Altogether, our two optimizations decreased the response time by four times than initial implementation. In Section 5.2, we will examine the performance of our RBC algorithm with and without these optimizations.

3.3 Aborting the Search Using Early Exit

The other vital half of the algorithm is the *early exit* for every MPI rank. If one of the ranks finds the key within the allotted time (or Hamming distance) limit, every other rank must gain knowledge that this event has occurred and the search must be subsequently aborted.

In distributed memory implementations, using a direct signal over shared memory is not possible. If each rank is to check if a change has occurred at the end of each iteration, this check would require network communication. The latency costs would accumulate with every computed key, which would be intractable for our large key search space. We outline two early exiting strategies and our exhaustive search approach below.

Exit-Allreduce: One approach is to periodically perform an `MPI_Allreduce` function call in MPI, which has every rank reduce every sub-answer to a singular one and redistribute this knowledge to every rank. In this context, after r keys are explored, each rank would call the `MPI_Allreduce` function based on the bitwise `OR` function over every rank's flag. If any one of the ranks' flag has been set, then all ranks would agree to set their flag. *In all that follows, we refer to this early exit strategy as* EXIT-ALLREDUCE. Unfortunately, while at smaller Hamming distances the overhead is minimal, at larger Hamming distances (and thus key spaces) even the largest r cannot scale with the exponential increase in load. As an illustrative example, assume first that the Hamming distance is 2 and that we have $p = 10$ ranks. This yields $\frac{\sum_{i=0} 2\binom{256}{x}}{10} \approx 3290$ keys per rank. If we set $r = 1000$ so that after every 1000 keys each rank stops and reduces, then each rank must only make 3 checks. If instead the search requires iterating up to a Hamming distance of 5, then $\frac{\sum_{i=0} 5\binom{256}{x}}{10 \cdot 1000} \approx 9.00 \times 10^5$ checks must be made.

Exit-Probe: Rather than forcing each rank to wait on each other at their synchronization points (as described above), the rank probes for communication from any source in a non-blocking way every r iterations. Once one of the ranks finds the key, this is communicated to all other ranks, where all ranks receive the probe signal using `MPI_Iprobe` and then execute the `MPI_Recv` function to unblock the sender. The communication overhead can be adjusted by changing the value of r before doing a non-blocking probe call similar to the EXIT-ALLREDUCE method above.

Exit-All Combinations: For completeness and to have a control for experiments, *the third early exit strategy is* EXIT-ALL COMBINATIONS, *where the algorithm disregards any notion of exiting the loop early.* Instead, it exhaustively

searches through every key under a given Hamming distance D. Even if a rank finds the key within D, every rank will completely iterate through the total key space at Hamming distance $\leq D$. Once a rank finds the key and the loop for Hamming distance D finishes, only a very minimal MPI call is needed to check at the end of every Hamming distance, limiting the amount of collective communication between ranks to D calls.

4 Implementation

To handle the distributed design, we chose `OpenMPI` in C as the foundation for the experiments for access to low-level bit manipulation, direct calls to the AES-NI instruction set, and the ability to coordinate the "early exit" strategy amongst all processes.

In a real world system, the key is randomly distributed amongst the ranks, and internally within rank's key space; there is equal chance that the key could be at the beginning or middle or end of a given rank's key space. However, the key should lie on average halfway through a random rank's key space. To find a statistical, dozens to hundreds of runs may need to be performed to find a suitable average response time that corresponds to the halfway point. To reduce the number of runs necessary to approach this average, a different, more direct method for generating i is used. We know that the average work needed to find the correct key is $\frac{\binom{256}{D}}{p}/2$, assuming that the exact Hamming distance D is known ahead of time. Therefore, in our experiments we presumed that the key resides exactly in the middle of 1 out of p ranks work; this in turn means that $i = \frac{R \cdot \binom{256}{D}}{p}/2$ for some random rank $R \in \mathbb{Z}_p^*$. The experiments in turn can be modified so that the random variable is picking a random rank R out of p uniformly. Once the random rank is chosen, we take the average of $\frac{R \cdot \binom{256}{D}}{p}$ and $\frac{(R+1) \cdot \binom{256}{D}}{p}$ and find our result for i in Equation (5).

$$i = \Big(\frac{R \cdot \binom{256}{D}}{p} + \frac{(R+1) \cdot \binom{256}{D}}{p}\Big)/2 = \frac{(2R+1) \cdot \binom{256}{D}}{2p} \tag{5}$$

5 Experimental Evaluation

For our experiments, several questions were considered. How many iterations (r) for each early exit methodology best reduces both communication and unnecessary key search overhead? Which early exit methodology is most efficient and thus provides the best scalability? What is the nature of the best method's scalability, and what factors may be influencing it?

5.1 Experimental Methodology

The RBC implementations were developed in C, targeting the Linux environment, and compiled using GCC with the `-O3` optimization flag and without any debug symbols.

While in a general purpose RBC engine the key may lie anywhere on the the combinatorial number line, for these experiments we chose the key placement specified in Section 4 to forcefully induce an expectation that the key is always in the middle of the number line. Specifically, the middle of this number line is the *average* case for a uniformly random key search. Using this approach allows less runs to be tested to gain a sufficient average, independent of the platform tested on.

For experiments that use the EXIT-ALL COMBINATIONS method of early exit, the key is also forced to be in the middle, but the exit mechanism is ignored. Instead, the algorithm merely reports that the key was found as a sanity check, and the rest of the number line is explored.

Our experiments are carried out on a compute cluster that contains up to 19 compute nodes of 2× Intel Xeon E5-2680 v4 (Broadwell) CPUs, each CPU with 14 physical cores with a base clock of 2.4 GHz and making up a total of 28 physical cores per node. We refer to this as PLATFORM A. In addition, each node of PLATFORM A contains 128 GiB of memory. The experiments presented in Section 5.3 used a single node of PLATFORM A with 28 total cores. All of the benchmarks in Section 5.4 and Section 5.5 were run with dedicated resources. All powers of 2 from 1 to 512 core benchmarks were tested 10 times.

For our experiments in Section 5.4 and Section 5.5, we define the parallel speedup to be the response time divided by the response time for a single core. Specifically, the single core response time uses the same MPI-based implementation as before, but will communicate with itself where necessary. Likewise, the parallel efficiency is the speedup divided by the rank count.

5.2 Comparing Our Custom Implementations vs. OpenSSL and GMP

A major factor in optimizing our algorithm is our use of low-level AES-NI instructions and our custom 256-bit unsigned integers as described in Section 3.2. We demonstrate the performance impact each component has on our algorithm in Table 1. Here, four implementations are tested, one for each combination of custom AES-NI vs. the OpenSSL library and custom fixed 256-bit unsigned integers vs. the GMP library. These implementations were tested using 28 cores of a single node of PLATFORM A with the EXIT-ALL COMBINATIONS method to demonstrate the performance of an exhaustive search. As shown in Table 1, with both libraries the runtime is 210.9 s, whereas using both of our optimizations yields 48.38 s and a speedup of 4.36×. If our optimizations do not suffer performance degradation from memory management operations, then the pure key search runtime should be an equivalent fraction in all implementations. That is to say, the runtime of `AES-NI+GMP` and `OpenSSL+uint256` share the same fraction of runtime as `Custom (AES-NI+uint256)`. Therefore, the addition of these two runtimes and subtracting one runtime of AES-NI+uint256 should add up to the total runtime of `OpenSSL+GMP`. This is proven by our experimental data, where $125.4 + 134.0 - 48.38 = 211.02$ which is approximately

Table 1: Response times (s), overall key throughput (keys/s) and speedup over the baseline OpenSSL+GMP implementation compared to our custom AES-NI+uint256, AES-NI+GMP, and OpenSSL+uint256, implementations. Speedup is defined to be the implementation runtime divided by the OpenSSL+GMP runtime.

Implementation	Response Time (s)	Key Throughput (keys/s)	Speedup
Custom (AES-NI+uint256)	48.38	1.82×10^8	4.36
AES-NI+GMP	125.4	7.06×10^7	1.68
OpenSSL+uint256	134.0	6.61×10^7	1.57
OpenSSL+GMP	210.9	4.18×10^7	1

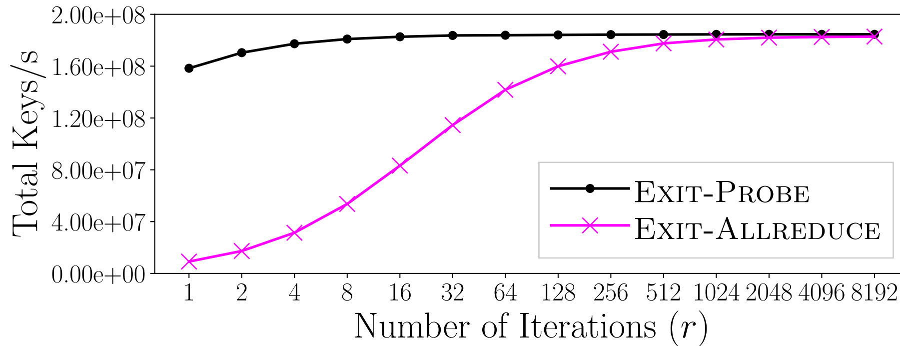

Fig. 2: The overall key throughput of the algorithm vs. the number of iterations (r) before communication is done comparing Exit-Probe and Exit-Allreduce.

OpenSSL and GMP = 210.9. This shows that the new codebase has absolutely removed all overhead that OpenSSL and GMP contain.

5.3 Performance Tuning of the Early Exit Iteration Parameter r

As both the Exit-Probe and Exit-Allreduce approaches of early exit rely on MPI communication, care must be given to optimizing the number of MPI calls made. There is a risk of spending too much communicating vs. allowing the rank to test more keys than are necessary. We experimented this relationship in Figure 2, by testing out the number of iterations between each MPI call (either MPI_Iprobe and MPI_Recv when found for Exit-Probe, or MPI_Allreduce for Exit-Allreduce). The number of iterations (r) used are all powers of 2 from 1 to 8192 inclusively to take advantage of modulo checks (i.e., $0 \overset{?}{\equiv} \text{count} \pmod{r}$).

There is a visible overhead at the smallest r in Figure 2, best showcasing the problem of communication overhead. For Exit-Probe, the key throughput plateaus at approximately $r \geq 128$ iterations. Likewise for Exit-Allreduce, the synchronous nature of MPI_Allreduce has a much greater impact, where the

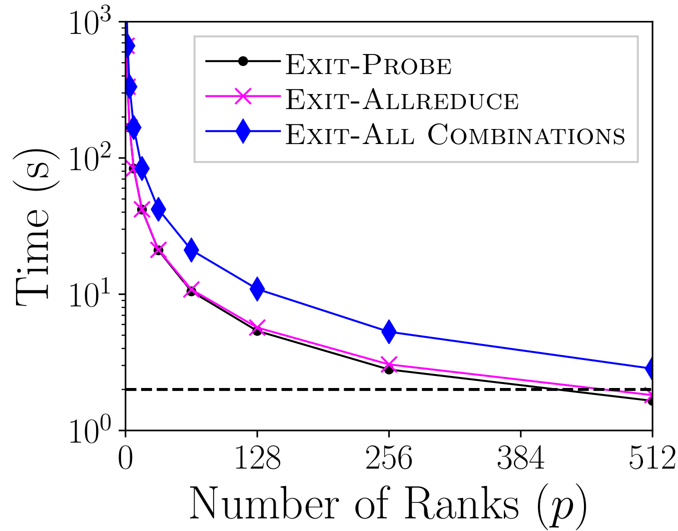

Fig. 3: Comparing Exit-All Combinations, Exit-Probe, and Exit-Allreduce as the response time of the algorithm vs. the number of ranks (p). The $T = 2$ s threshold is also shown as the horizontal, dashed line.

two methods do not converge until about 4096 iterations. We observe a key rate of 185 million keys per second (for 28 cores), the 8192 iterations has minimal effect on increasing latency and reading more keys than are necessary.

5.4 Evaluation of Early Exit Strategies

For the next experiment, the next question is which early exit method is more efficient and scalable between Exit-Probe and Exit-Allreduce. This experiment is set up by running all ranks/cores counts of powers of 2 from 1 to 512 inclusively on Platform A for the early exit methods Exit-Probe, Exit-Allreduce, and Exit-All Combinations.

In Section 5.3, it was shown that at minimum the recommended number of iterations for every MPI call(s) was 128 and 4096 for Exit-Probe and Exit-Allreduce, respectively. While anywhere within the range of $\sim 128 - 8192$ and $\sim 4096 - 8192$ within the tested parameters for the two methods respectively were sufficient, we chose **128** iterations for Exit-Probe and **8192** iterations for Exit-Allreduce for our experiments.

In Figure 3, all three methods are plotted as time against an increasing rank count. Exit-Probe and Exit-Allreduce show similar times, with Exit-Probe achieving the lowest response time, whereas Exit-All Combinations is on average twice as slow as either method as would be expected since it searches through *all* the keys without early exit.

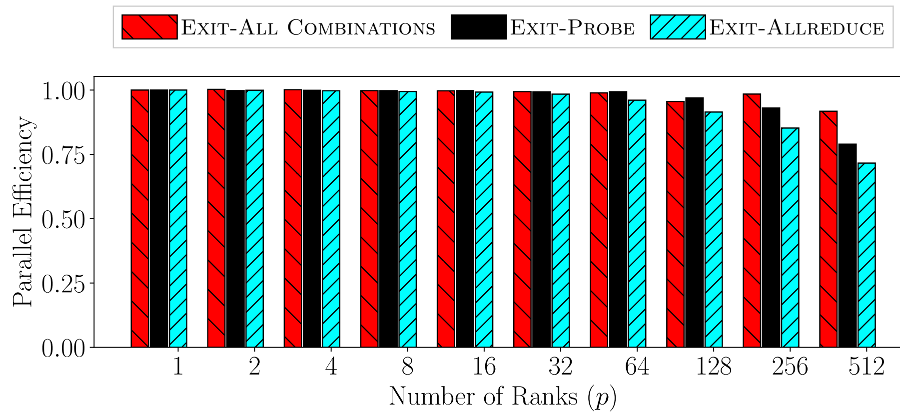

Fig. 4: Comparing Exit-All Combinations, Exit-Probe, and Exit-Allreduce as the response time of the algorithm vs. the parallel efficiency.

Figure 4 compares the early exit strategies as a function of parallel efficiency (as a percentage) against the number of ranks p. At $p \leq 32$ ranks, all three methods are effectively equivalent in their scalability, with none being anymore efficient than Exit-All Combinations. This is also what one would expect without MPI communication. Starting at $p = 64$ ranks, the efficiency of Exit-Allreduce begins to degrade, whereas Exit-All Combinations begins degrading at $p = 128$ ranks. By $p = 512$ ranks, Exit-All Combinations's parallel efficiency is 91.77%, Exit-Probe's is 78.97%, and Exit-Allreduce's is at 71.61%.

Note that even Exit-All Combinations loses parallel efficiency regardless of not using communication. At 512 ranks, the response times for Exit-All Combinations, Exit-Probe, and Exit-Allreduce are 2.84 s, 1.65 s, and 1.81 s respectively. As the runtime decreases, small variances in switching hardware and latency become a more significant proportion of time, adding equally significant variance to the total runtime as the number of ranks increases. These experiments show that while the algorithm scales sufficiently well up to 512 ranks/cores, there is still likely to be a law of diminishing returns with careful consideration for hardware when implementing RBC for distributed-memory clusters of compute nodes.

After considering all three early exit methods, Exit-Probe is the recommended as the most efficient up to 512 ranks. In Section 5.5, we will examine Exit-Probe more thoroughly.

5.5 Time-to-Solution and Scalability

As Exit-Probe is the best method chosen from Section 5.4, we explore a more in-depth analysis on its scalability and speedup. No new experiment was performed for this section, and instead the same data is used as in Section 5.4.

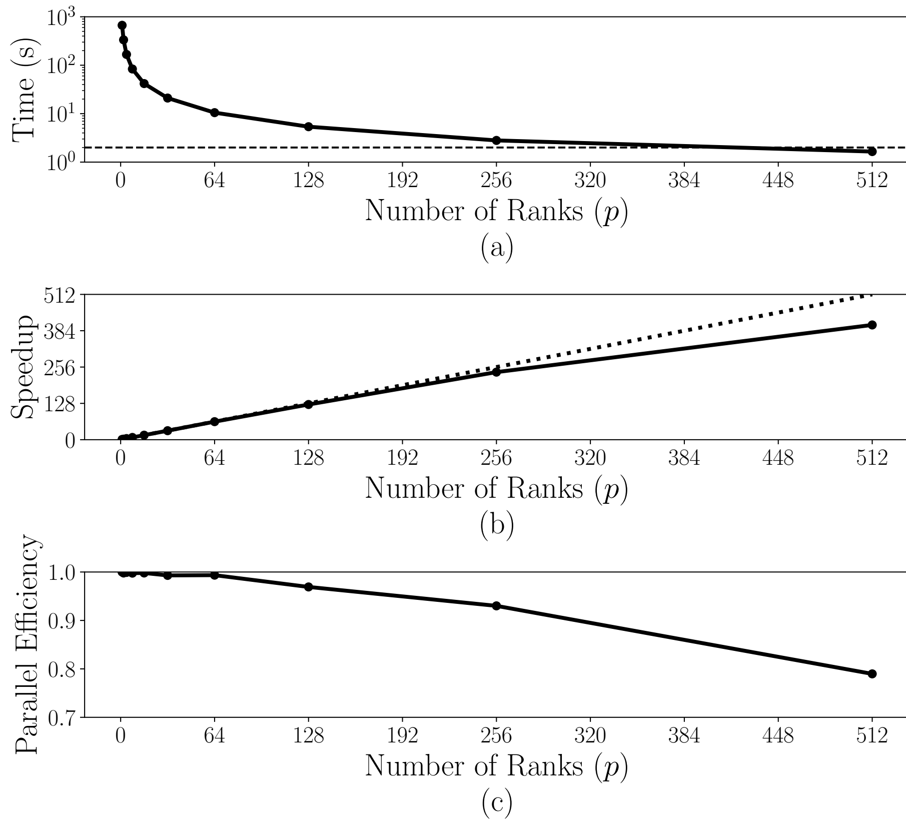

Fig. 5: (a) The response time of our algorithm vs. the number of ranks (p). (b) The parallel speedup vs. p, where the dashed line indicates a perfect speedup for comparison. (c) The parallel efficiency vs. p. All experiments are executed on up to 512 ranks on dedicated compute nodes. Experiments were carried out on Platform A.

We analyzed the response time, speedup and parallel efficiency vs. the number of ranks/cores to find K_A derived from K_B. As stated in the motivation of the paper (Section 1), our primary goal is to achieve a maximum latency of $T = 2$ s. Figure 5(a) plots the response time vs. the number of ranks/cores (p), where we find that we are able to successfully achieve authentication within $T = 2$ s when using $256 < p \leq 512$ ranks. With no greater precision than 256 and 512 ranks tested, we can only estimate that 384 ranks is the minimum required ranks on Platform A to meat the $T = 2$ s threshold. The response time starts at 667 seconds at 1 rank, roughly decreasing by a factor of 2 with each factor of 2 increase in ranks up to $p = 128$. With 128, 256, and 512 ranks, the response times are 5.39 s, 2.80 s, and 1.65 s respectively.

Figure 5(b)–(c) plot the parallel speedup and parallel efficiency vs. p. The algorithm achieves good scalability up to $p = 512$ ranks, where we achieve a speedup of 404×, corresponding to a parallel efficiency of 77.9%. Combined with Figure 4, both methods of early exit lead to a similar drop in efficiency. Regardless of synchronous or asynchronous communication, this indicates that the communication required to notify other ranks when the key has been found is non-negligible. Additionally, since all ranks independently search the key space, the ranks will compute more keys than needed since there is a delay between when the key is found by a rank, and when the signal is received to stop the search. As the number of ranks increase, there is a greater total delay when communicating that the search needs to terminate.

6 Conclusions and Future Work

Response-based cryptographic engines have the potential to be applicable in many different use cases. The disparity of compute power between client devices, servers, and compute clusters is intractible. Past research has focused on placing more computational pressure on the client-side by developing complex error-correcting methods. Before, the concept of taking advantage of the massive compute power of modern-day clusters was only illustrated in theory [9, 12, 14].

Through our experiments, we took the next step by displaying the capabilities of the protocol in practice. We developed a novel methodology for the server to iteratively search through the client device's surrounding key space with increasing Hamming distances. By utilizing the combinatorial number system and an efficient early exit strategy, we were also capable of massively distributing the search problem up to 512 cores.

Future work to consider are exploring other symmetric and asymmetric encryption schemes, such as elliptic-curve cryptography. Furthermore, other recent and novel post-quantum cryptographic encryption schemes, namely lattice-based and hash-based cryptography, should be considered. Another direction to consider is using graphics processing units (GPUs) for general purpose computation. The GPU's high computational throughput may further improve performance.

As discussed in Section 5.5, the networking of the platform can have a significant impact on the overall throughput of the implementation as the core count increases.

References

1. Anderson, S.E.: Bit Twiddling Hacks (2005), http://graphics.stanford.edu/~seander/bithacks.html
2. Antoniadis, A., Sklavos, N., Kavun, E.B.: An efficient implementation of a delay-based puf construction. In: proceedings of Trustworthy Manufacturing and Utilization of Secure Devices Workshop, Design, Automation and Test in Europe Conference,(DATE'20), Paris, France (2020)

3. Assiri, S., Cambou, B., Booher, D.D., Ghanai Miandoab, D., Mohammadinodoushan, M.: Key Exchange using Ternary system to Enhance Security. In: IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC). pp. 0488–0492 (2019)
4. Booher, D.D., Cambou, B., Carlson, A.H., Philabaum, C.: Dynamic Key Generation for Polymorphic Encryption. In: IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC). pp. 0482–0487 (2019)
5. Bösch, C., Guajardo, J., Sadeghi, A.R., Shokrollahi, J., Tuyls, P.: Efficient Helper Data Key Extractor on FPGAs. In: Oswald, E., Rohatgi, P. (eds.) Cryptographic Hardware and Embedded Systems. pp. 181–197. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
6. Boyen, X.: Reusable Cryptographic Fuzzy Extractors. In: Proceedings of the 11th ACM Conference on Computer and Communications Security. pp. 82–91. CCS '04, ACM, New York, NY, USA (2004)
7. Cambou, B., Telesca, D.: Ternary Computing to Strengthen Information Assurance. Development of Ternary State based Public Key Exchange. In: IEEE, SAI Computing Conference (2018)
8. Cambou, B.: Physically Unclonable Function Generating Systems and Related Methods (May 29 2018), US Patent 9,985,791
9. Cambou, B.: Unequally Powered Cryptography with Physical Unclonable Functions for Networks of Internet of Things Terminals. In: Proceedings of the Communications & Networking Symposium. pp. 4:1–4:13. CNS '19, Society for Computer Simulation International, San Diego, CA, USA (2019), http://dl.acm.org/citation.cfm?id=3338063.3338067
10. Cambou, B., Flikkema, P.G., Palmer, J., Telesca, D., Philabaum, C.: Can Ternary Computing Improve Information Assurance? Cryptography 2(1) (2018), https://www.mdpi.com/2410-387X/2/1/6
11. Cambou, B., Orlowski, M.: Puf designed with resistive ram and ternary states. In: Proceedings of the 11th Annual Cyber and Information Security Research Conference. pp. 1–8 (2016)
12. Cambou, B., Philabaum, C., Booher, D.: Response-based Cryptography with PUFs, NAU Case D2018-049
13. Cambou, B., Philabaum, C., Booher, D.: Replacing error correction by key fragmentation and search engines to generate error-free cryptographic keys from pufs. CryptArchi (2019)
14. Cambou, B., Philabaum, C., Booher, D., Telesca, D.A.: Response-Based Cryptographic Methods with Ternary Physical Unclonable Functions. In: Arai, K., Bhatia, R. (eds.) Advances in Information and Communication. pp. 781–800. Springer International Publishing, Cham (2020)
15. Chen, A.: Comprehensive assessment of rram-based puf for hardware security applications. In: 2015 IEEE International Electron Devices Meeting (IEDM). pp. 10.7.1–10.7.4 (2015)
16. Chen, B., Ignatenko, T., Willems, F.M.J., Maes, R., van der Sluis, E., Selimis, G.: A Robust SRAM-PUF Key Generation Scheme Based on Polar Codes. In: IEEE Global Communications Conference. pp. 1–6 (2017)
17. Delvaux, J., Gu, D., Schellekens, D., Verbauwhede, I.: Helper Data Algorithms for PUF-Based Key Generation: Overview and Analysis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 34(6), 889–902 (June 2015)
18. Gassend, B., et al.: Silicon Physical Eandomness. In: Proceedings of the 9th ACM Conference on Computer and Communications Security. pp. 148–160 (2002)

19. Guajardo, J., Kumar, S.S., Schrijen, G.J., Tuyls, P.: FPGA Intrinsic PUFs and Their Use for IP Protection. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2007. pp. 63–80. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
20. Gueron, S.: Intel® Advanced Encryption Standard (AES) New Instructions Set. Tech. Rep. 323641-001, Intel Corporation (May 2010), https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf
21. Harary, F., Beckenbach, E.: Applied Combinatorial Mathematics (1964)
22. Herder, C., Yu, M., Koushanfar, F., Devadas, S.: Physical Unclonable Functions and Applications: A Tutorial. Proceedings of the IEEE 102(8), 1126–1141 (Aug 2014)
23. Hiller, M., Merli, D., Stumpf, F., Sigl, G.: Complementary IBS: Application Specific Error Correction for PUFs. In: 2012 IEEE International Symposium on Hardware-Oriented Security and Trust. pp. 1–6 (June 2012)
24. Hofer, M., Boehm, C.: An Alternative to Error Correction for SRAM-Like PUFs. In: Mangard, S., Standaert, F.X. (eds.) Cryptographic Hardware and Embedded Systems, CHES 2010. pp. 335–350. Springer Berlin Heidelberg, Santa Barbara, California (Aug 2010)
25. Holcomb, D.E., Burleson, W.P., Fu, K., et al.: Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags. In: Proceedings of the Conference on RFID Security. vol. 7, p. 01 (2007)
26. Kang, H., Hori, Y., Katashita, T., Hagiwara, M., Iwamura, K.: Cryptographie Key Generation from PUF Data Using Efficient Fuzzy Extractors. In: 16th International Conference on Advanced Communication Technology. pp. 23–26 (Feb 2014)
27. Keller, C., Gürkaynak, F., Kaeslin, H., Felber, N.: Dynamic memory-based physically unclonable function for the generation of unique identifiers and true random numbers. In: 2014 IEEE International Symposium on Circuits and Systems (ISCAS). pp. 2740–2743. IEEE (2014)
28. Knuth, D.E.: The Art of Computer Programming: Vol. 4, No. 1: Bitwise Tricks and Techniques-Binary Decision Diagrams. Addison Wesley Professional (2009)
29. Knuth, D.E.: Generating All Combinations and Partitions. Addison-Wesley (2010)
30. Maes, R., Tuyls, P., Verbauwhede, I.: A Soft Decision Helper Data Algorithm for SRAM PUFs. In: 2009 IEEE International Symposium on Information Theory. pp. 2101–2105 (June 2009)
31. Maes, R., Verbauwhede, I.: Physically Unclonable Functions: A Study on the State of the Art and Future Research Directions, pp. 3–37. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
32. Mohammadinodoushan, M., Cambou, B., Philabaum, C., Hely, D., Booher, D.: Implementation of Password Management System Using Ternary Addressable PUF Generator. In: IEEE SECON 2019: IEEE STP-CPS Workshop (Jun 2019)
33. Prabhu, P., Akel, A., Grupp, L.M., Yu, W.K.S., Suh, G.E., Kan, E., Swanson, S.: Extracting device fingerprints from flash memory by exploiting physical variations. In: McCune, J.M., Balacheff, B., Perrig, A., Sadeghi, A.R., Sasse, A., Beres, Y. (eds.) Trust and Trustworthy Computing. pp. 188–201. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
34. Rahman, M.T., Rahman, F., Forte, D., Tehranipoor, M.: An aging-resistant ro-puf for reliable key generation. IEEE Transactions on Emerging Topics in Computing 4(3), 335–348 (2016)

35. Rožić, V., Yang, B., Vliegen, J., Mentens, N., Verbauwhede, I.: The monte carlo puf. In: 2017 27th International Conference on Field Programmable Logic and Applications (FPL). pp. 1–6. IEEE (2017)
36. Schrijen, G.J., van der Leest, V.: Comparative Analysis of SRAM Memories Used As PUF Primitives. In: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 1319–1324. DATE '12, EDA Consortium, San Jose, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2492708.2493033
37. Sutar, S., Raha, A., Raghunathan, V.: D-puf: An intrinsically reconfigurable dram puf for device authentication in embedded systems. In: 2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES). pp. 1–10. IEEE (2016)
38. Taniguchi, M., Shiozaki, M., Kubo, H., Fujino, T.: A Stable Key Generation from PUF Responses with a Fuzzy Extractor for Cryptographic Authentications. In: IEEE 2nd Global Conference on Consumer Electronics (GCCE). pp. 525–527 (2013)
39. Vatajelu, E.I., Natale, G.D., Barbareschi, M., Torres, L., Indaco, M., Prinetto, P.: Stt-mram-based puf architecture exploiting magnetic tunnel junction fabrication-induced variability. J. Emerg. Technol. Comput. Syst. 13(1) (May 2016), https://doi.org/10.1145/2790302
40. Yu, M., Devadas, S.: Secure and Robust Error Correction for Physical Unclonable Functions. IEEE Design Test of Computers 27(1), 48–65 (Jan 2010)