[Albert Carlson, Benjamin Williams et al. (Oct. 20, 2025). Calculating the Increased Speed of Polymorphic Encryption. 2025 IEEE 16th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON), DOI: 10.1109/UEMCON67449.2025.11267551. IEEE. Source: https://ieeexplore.ieee.org/document/11267551]

Calculating the Increased Speed of Polymorphic Encryption

Albert Carlson[*], **Benjamin Williams**[†], **Sai Ranganath Mikkilineni**[‡], **Christopher Briscoe**[§], and **Mandeep Singh**[¶]
[*]Computer Science Department, National University, San Diego, CA, USA.
[†]Computer Science Department, University of Idaho, Moscow, ID, USA.
[‡]College of Business, Delaware State University, Dover, DE, USA.
[§]Department Chair of Mathematics and Physics, Emet Classical Academy, New York, NY, USA.
[¶]Independent Researcher.
Email: [*]acarlson2@nu.edu, [†]will9847@vandals.uidaho.edu, [‡]SMikkilineni@desu.edu, [§]Cbriscoe@emetclassicalacademy.org, [¶]msingh9001@gmail.com

***Abstract*—Polymorphic encryption techniques are relatively new and generally not well understood. Although practitioners claim that this type of encryption is slower due to steps such as sharding, selecting random keys, and choosing random ciphers, these encryptions can be even faster than the linear version of encryption for the same message. However, there has been no published mathematical analysis of this claim. (Security analysis is beyond the scope of this paper.) The study presented in this paper encompasses shards of both identical and differing sizes. It also demonstrates that partitioning and queuing functions can be handled by one or more cores, enabling efficient and predictable operation and analysis. Logic might dictate that speeding up processing would require the use of multiple cores, threads, or GPUs, since shards constitute orthogonal encryption problems. However, with only two cores, a polymorphic cipher offers superior security without loss of speed. Ambiguous statistical analysis techniques, such as benchmarking, are not used due to the impossibility of effectively controlling for all hardware variations and configurations, and because even on the same hardware, non-real-time OS schedulers can cause significant variation from one test to the next. Rigorous mathematical analysis is used here, because it is much more reliable for comparative study.**

***Keywords*—Polymorphic cipher, encryption, parallel processing, threads, polymorphic engine.**

I. Introduction

Polymorphic encryption is a relatively underutilized encryption methodology. Its users claim that these ciphers are at least as fast, if not faster, and more secure than the more commonly used symmetric key and asymmetric key algorithms. No proofs have previously been presented to support these assertions. The lengthy proofs of these claims will need to be addressed separately. This paper examines the claim that polymorphic ciphers can reduce the time required for encryption and decryption, considering both cases of identical and variable-sized shards. It also gives the conditions necessary for subsuming overhead and latency in the operation of the ciphers. The assertion of superior security will be addressed in another paper.

A. Organization of the Paper

This paper consists of four sections. The first section is the Introduction, where the problem is outlined. In Section II, essential background information is provided that is required as the basis for the mathematical description of the implementation of a polymorphic cipher. After laying out the background, the third section presents our analysis, which includes the mathematical foundation of the polymorphic cipher. We also demonstrate that the overhead required for polymorphic encryption can be mitigated by utilizing cores, threads, and/or GPUs, resulting in a reduction in the time needed for encryption compared to the linear application of a cipher. We conclude our paper in Section IV, where we summarize our results and discuss how the work can be extended in the future.

II. Background

This section presents the key background concepts necessary to demonstrate the increased speed of Polymorphic Encryption Engines (PMEs) [1] algorithms. The paper discusses Polymorphic RNGs, shards, rounds, and parallelization of processing encryption streams.

A. Polymorphic RNGs

RNGs are programs that generate a random series of numbers in response to each request for such a number from a user or program. There are various types of RNGs, ranging from true RNGs [2] to pseudo-random RNGs (PRNGs) [3]. Most RNGs are PRNGs due to the challenges in creating truly random sequences [4]. RNGs also have varying strengths [5], [6]. The strongest are known as "cryptographic strength" RNGs or CSRNGs [7]. These RNGs are characterized by long periods (maximal λs) and complex sequences. There are also many different types and technologies of RNGs. The selection of the proper kinds of RNGs is the responsibility of the cipher designer.

A true RNG (TRNG) gives a verifiably random number [8] for each access [9] and has no pattern. However, no computer

can generate a true random sequence [10]. Computer-generated RNGs are pseudo-random (PRNG) and follow patterns such that

$$f(n+1) = f(f(n)) \tag{1}$$

which cycles in a period of λ accesses. There may be more than one cycle for a PRNG, which means that for each cycle of the PRNG (λ_i)

$$\lambda_i \leq 2^{|B|} \tag{2}$$

where $|B|$ is the number of bits in the representation of the calculated number. A "maximal cycle" occurs when there is a single cycle for the PRNG where $\lambda = 2^{|B|}$.

The numbers in a cycle are unique and only repeat when the cycle is completed. The succeeding numbers then repeat in the same order as the preceding cycle. Therefore, the cycle limits can be detected when $f(n) = f(n+\lambda)$. Each PRNG has a unique sequence that can be recognized and predicted after ρ [11] accesses. For the event with the most complicated RNG sequences, ρ is often small and constitutes a signature for the particular PRNG.

Random number generators (RNGs) play a crucial role in polymorphic encryption. Often, these RNGs are themselves polymorphic ("polyRNGs") [12]. They are used to select both the cipher and the key used in the encryption process. Depending on the chosen RNG for this process, the RNG can comprise a significant portion of the overhead costs and act as a linear bottleneck. Therefore, understanding polyRNGs is essential.

Recent advances in polyRNGs have resulted in an RNG based on the Geffe generator [13] model that has many of the characteristics of a TRNG in a PRNG that can be extended to any size. These polyRNGs are based on the polymorphic encryption model [14], [15], which is both fast and secure. Comprising multiple RNGs combined through a serial algorithm, polyRNGs can be implemented in hardware, software, or as a hybrid unit. It is possible to create very long serial sequences with the constituent RNGs being changed according to TTL principles, thereby constantly mixing the selected RNGs in the same manner as polymorphic cipher/key pairs. This methodology prevents the attacker from targeting the ρ of the ciphers, thereby avoiding an attack on the cipher through its RNG [16].

The RNG is the linear bottleneck in speed. Shard sizes can be adjusted within their unicity distance [17] to fine-tune timing to reduce or eliminate the impact of the RNG bottleneck. It is essential to note that the unicity distance of any cipher and the strength of a polymorphic cipher are directly dependent on the encryption algorithm selected and used. The greater the unicity distance, the less often the cipher/key pair needs to be changed. Given the greater unicity distance, the time to live (TTL), or time until the cipher/key pair of the shard must be changed, and the corresponding shard size can also be larger, requiring fewer RNG accesses. A reduced number of accesses means a reduced impact of the RNG bottleneck and less time and resources used during the encryption process.

With fast processors and strong algorithms that have long unicity distances, it should be possible to reduce the RNG bottleneck to the point where the data storage and transmission are the only significant bottlenecks. Verifying this assertion still requires testing. Analysis is also necessary to theoretically determine the optimal shard sizes to maximize encryption speed with these shards.

B. Shards

PMEs [15] use other ciphers, and irregularly change the ciphers used, for portions of encrypted messages ($|M|$), where

$$|S_i| \leq |M| \tag{3}$$

and

$$|M| = \sum_{i=1}^{i_{max}} |S_i| \tag{4}$$

where i_{max} is the number of shards in the message and $|S_i|$ is size of the i^{th} shard in the message. These sub-messages are termed "shards" [15] because of the image of a broken pane of glass, representing the message being broken into irregular pieces that correspond to grouping characters in the message into sub-messages. Therefore, a shard can be thought of as a smaller portion of the whole message treated as a message on its own. The size of the shard is limited by the entropy and unicity distance (n) [17]. A shard (S_i) has a size

$$1 \leq b_i < n_i \tag{5}$$

and the block is some multiple of the block size (B) of the encrypted character. That is,

$$b_i \,\%\, B = 0 \tag{6}$$

Shards do not have to be of identical size, although they may be the same size. A shard also typically requires a "randomly" selected cipher/key pair used to encrypt that shard [18].

C. Threading Overhead

Parallelization in computing is not free. Several factors influence the magnitude of gains that can be achieved by distributing a process across multiple threads. Even on a system with many CPU cores, each thread takes CPU time to set up and begin execution, and each thread consumes a certain baseline amount of memory for information that the OS needs to manage it. This creates a static cost for each thread, which can diminish the gains of parallelization as the number of threads increases.

There are two common types of threads, I/O bound and CPU-bound [19]. I/O-bound threads are those that spend a significant amount of time blocked, waiting for slow I/O operations to complete. CPU-bound threads are those that are primarily limited by available CPU time. While a system may be able to efficiently handle many more I/O bound threads than it has CPU cores, the general rule is that a process should have no more CPU-bound threads than the number of CPU cores available, because threads that are waiting for other threads to

finish are consuming memory and OS management resources without increasing the rate of computation.

Encryption threads are CPU-bound. This means that increasing the number of encryption threads only provides performance improvements up to the limit of the number of threads the CPU can run in parallel. Simple CPU architectures can only run one thread per CPU core. However, some more advanced architectures can execute two or more threads on a single core (Intel Hyper-Threading at two threads per core [20], SPARC T series processors at up to 8 threads per core [21]), typically at reduced performance for each thread but with an overall performance gain through increased computation rate. For the scope of this discussion, CPUs can be treated as having many virtual cores, equal to the number of physical cores multiplied by the number of threads per core. For example, the Intel Core i7-8700 has six physical cores that can run two threads per core, giving it 12 virtual cores and making it capable of running up to 12 CPU-bound threads efficiently [22]. If a 13th CPU-bound thread is added, that thread will share execution time with the other threads, causing an increase in thread overhead costs to memory and execution without improving performance.

Figure 1 graphs the execution of a polymorphic encryption on a system that can run four threads in parallel. The main thread is running on the first core and handles dispatch, performs shard bookkeeping, and dispatches threads as needed. The other three threads are encrypting shards. Each block in thread 0 represents a dispatch event, where a shard key is generated and a new shard is created and dispatched to run in another thread. The block at the very top creates shard zero and sends it to thread 1 for encryption. The next block creates shard one and so on. After the third block, there is a gap because all of the cores are currently working, so the main thread has to wait for one of the shards to be completed to dispatch the next one. It might be tempting to add another thread to help fill in those gaps, but doing that could delay the main thread, leaving other threads idle for some time after they are finished. If the dispatch time is shorter and the gap is large enough, the gains from adding a fourth shard thread might pay off. This can be adjusted by tuning the shard size, which may result in a potential reduction in overall security. It might be tempting to add additional shards beyond that, to ensure that there's always a shard thread queued to replace the next one that is finished immediately. Still, once the number of threads exceeds the number of virtual cores, each additional thread will require the OS to spend more time handling scheduling without providing any performance improvement, which will slow down the process rather than increasing its speed. In the ideal case, the data being processed would be split into several shards exactly equal to the number of virtual cores, they would all be dispatched, and then the main thread would block until they had all completed, but basing shard size exclusively on speed optimization would dramatically reduce the security of the result. Therefore, it is necessary to choose a shard size that is suitable for security and then distribute the shards in a way that keeps the CPU busy without excessive thread overhead, thereby avoiding a net performance loss.

In more concrete terms, the total gains obtained from multithreading can be roughly calculated using equation 7, where n is the number of threads running at one time, s is the total number of shards, t is time, and v is the number of virtual cores being used. (Note that n cannot be lower than v, because even if the machine has more than n cores, if there are only n threads running, only n cores are being used.)

$$t_{real} = ((t_{overhead} * n + t_{shard}) * s)/v \tag{7}$$

Note that overhead costs scale with the number of running threads and are multiplied by the total number of shards. This is because the overhead cost persists throughout the entire process, the length of which is determined by the total number of shards. Increasing n without increasing v will always increase the amount of real time required to complete the encryption.

In real-world applications, other factors can influence overhead, such as memory and disk latency, which may yield small gains when adding one or two additional threads beyond the number of virtual cores. Still, there is so much variation in hardware that it is impossible to account for this entirely. On the other hand, too many threads can create cache coherency issues that cause the program to slow down. For ideal optimization, test-driven fine-tuning is necessary, which is far beyond the scope of this paper.

D. Rounds

Assume that there are c processing cores or general processing units (GPUs) [23] in a computer. The set of shards that can be processed in parallel is referred to as a "round" of processing. Each round processes c shards simultaneously, potentially significantly reducing the encryption time — the greater the value of c, the more shards that can be processed simultaneously.

E. Parallelization of Processing Encryption Streams

Processors or computers with multiple cores can be used to run different programs, or the same program with various inputs, on each core. Many processors now have multiple cores or threads that are typically unused during the normal course of running programs [24]. One of the reasons they often remain unused is that it is generally tricky to synchronize program segments that require effective programming [25]. However, orthogonal problems [26] do not require such synchronization and can be treated as different, independent programs. Each core, thread, or GPU must be loaded separately and then allowed to run its program, with each core loaded consecutively. Such an action is shown in Figure 1. The controlling or the "overhead" program can continuously fill the queues of each of the other cores. There are two phases of operation:

1) Initially filling the queues - Each of the processing cores is filled sequentially. The queues can be filled during an overhead period, or can be filled as each shard in a

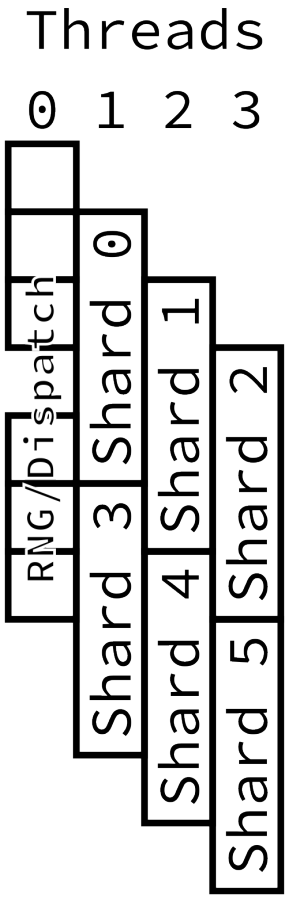

Fig. 1. Polymorphic Threading Model

queue is encrypted and transmitted. Resolving all queues simultaneously creates memory issues, as a queue for each core must be created and populated. The size of the queue will vary, but can be calculated on average. If there are c cores and the average size of a shard is

$$|\overline{S_i}| = \frac{|M|}{i_{max}} \tag{8}$$

where $|M|$ is the message size, in bytes, and i_{max} is the number of shards in the message. Then the queue size for a core ($|Q|$) will, on the average, be

$$|Q| = \frac{(|\overline{S_i}|)(i_{max})}{c} \tag{9}$$

Loading the queues in this manner may not be the most efficient method of operation. A second method is to load each core with a single shard at a time, with a queue being prepared as the shard being serviced is encrypted and transmitted. In this case, one or more cores/threads are used to keep the other cores busy with data to be encrypted. Assume that only a single core is needed to service the rest of the cores. First, the encryption process begins by filling each core queue. This period is represented by overhead and latency, denoted as $t_{O,L}$.

2) Queues remain full - All cores are fully loaded and remain so until the shards are completely loaded for processing.

III. Analysis

For reference in understanding the overarching process of simple polymorphic encryption, refer to the following. (Trivial bookkeeping steps are omitted, and this implementation assumes that all of the encrypted data will fit in available memory.)

1) Initialize the shard key generator using the master key. This happens on the Main Thread.
2) Load one shard of data from the data stream.
3) Generate a key for the shard.
4) Send the shard and key to a thread for encryption. This initiates Shard Thread(s).
   a) Encrypt the shard using the provided key.
   b) Store the encrypted data.
5) Repeat steps 2 to 4 until the end of the stream.
6) Combine the encrypted shards in their original order.
7) Store the combined ciphertext.

Note that there is no nonce/offset, initialization vector, or other data is required, as this is not necessarily an AES mode-based encryption (and thus it should be evident that it is also not vulnerable to attacks that AES modes are vulnerable to, like meet-in-the-middle attacks).

A. Number of Shards

Let the number of shards be denoted by i_{max}. The number of shards is determined by the size of the message and the size of each shard. The size of the shard will be represented by $|S_i|$. There are practical limits on the number of shards. Since each shard is the same size, except perhaps the last shard. Then the condition $|M| \ \% \ |S_i| \leq |S_i| - 1$ must be true, although a perfect division of the message is desired. Therefore,

$$i_{max} = \frac{|M|}{|S_i|} \tag{10}$$

Shards can vary in size from $|S_i| = |M|$ for a single shard to $|S_i| - 1$ at the minimum size but maximum number of shards. The question of shard size is then made based on the cipher to be used and the block size of the PT data for that cipher. Ideally, S_i would be chosen based on the number of processing units available to the computer, either cores or GPUs. Note that if the number of cores, $c << i_{max}$, then the terms for loading and completion may be ignored.

The number of shards that are encrypted at the beginning of the action has previously been characterized. During the encryption process, the cores are entirely filled. Calculating the time required during this phase of encryption depends on the number of shards remaining. The number of shards during this phase (i') is given by eliminating the number of shards that are encrypted at the beginning (i_b) and end (i_e) from the completed number of shards in the message (see Figure 2).

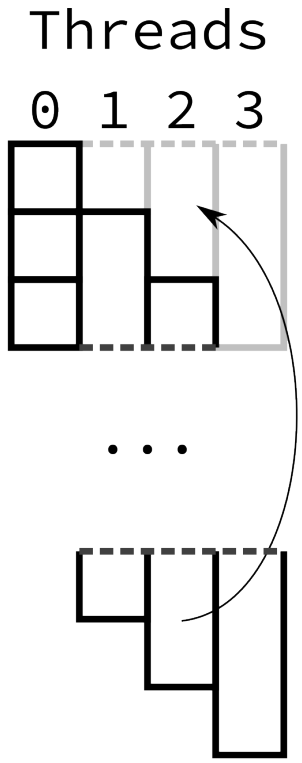

Fig. 2. Polymorphic Thread Interleaving

B. Cases of Interest

We examine three potential scenarios within the parameters of where PMEs are utilized, whether they are faster, and the conditions required for PMEs to achieve this speed. These cases are:

1) The case of Identical-Sized Shards and Small Overhead: In this case, we consider if the PME has shards of identical sizes, with the calculation and control overhead smaller than the time required to encrypt a single shard. Let the time to encrypt a single block be K. Then the time required to encrypt a message (t_e) of size $|M|$ in linear fashion is

$$t_e = K|M| \tag{11}$$

By "linear fashion," we mean a single core/thread is used to encrypt one block at a time and transmit them in that order. This is the traditional way of effecting encryption. For example, using the Advanced Encryption Standard (AES) with Cipher Block Chaining (CBC) mode [27] requires working on a block of plain text (PT), completing the encryption, and then either concatenating that block to the encrypted text stream or transmitting it. However, often this type of data also requires information from one or more preceding blocks or an initialization vector (IV). Ciphers with this type of dependency must be encrypted linearly, one block at a time, to ensure all the required input for encryption is available to complete the process.

Some ciphers do not rely on data from previously encrypted blocks, such as the substitution (S) cipher. In this case, each block can be presented independently, meaning that the time for encryption is the same for both encryptions if, and only if, the block sizes are identical.

The key to increasing speed is utilizing parallel cores, threads, or GPUs. For such parallelization to work, each shard must be orthogonal and independent of every other shard. With this condition, the problems can be solved simultaneously. Shards of the same size will take the same time to process, meaning that in the same amount of time, the number of blocks that can be encrypted in parallel will be c blocks, where c is the number of cores, threads, or GPUs assigned to the task. First, the number of shards in the message must be calculated.

a) Number of Rounds: The number of rounds depends on the number of cores or GPUs. That number, r, is minimized when all available cores are utilized in the encryption process. The minimum number of rounds required (r_{min}) is given by

$$r_{min} = \frac{i'}{c}, \text{ where } c \geq i' \tag{12}$$

However, most computer chips have a limited number of cores or GPUs. Due to the limited availability of multi-core capabilities and the rarity of parallel programming solutions [28], it is unlikely that the number of cores will exceed the number of shards for most messages. The cost of additional cores also restricts the number of available parallel paths that can be used. Furthermore, as the size of the message increases, it becomes less likely that there will be sufficient cores or GPUs to enable encryption in a single round. This situation is also complicated by the need for a program to control and submit shards to the computer for encryption. At least one core must be reserved for scheduling, breaking the message into shards, and loading or offloading the data. Therefore, the number of rounds for this case is

$$|r| = \frac{i'}{c-1} \tag{13}$$

Therefore, the total processing time is

$$t_{e,p} = |r|K \tag{14}$$

and the time saved is

$$t_e - t_{e,p} = |M|K - |r|K = K(|M| - |r|) \tag{15}$$

Assume that the overhead P is fast enough to set up for the next round. Then the number of final rounds ($|r_f|$) is given by

$$|r_f| = |r| + 2(c-1) \tag{16}$$

b) Overhead Requirements: In this case, the overhead processing takes no more time than encrypting a shard. That is, the overhead must be

$$P \leq K|S_i| \tag{17}$$

To save time on the encryption of the message

$$K|M| \geq K|r||S_i| \geq \frac{K(i_{max})|S_i|}{c-1} \tag{18}$$

$$|M| \geq \frac{(i_{max})|S_i|}{c-1} \tag{19}$$

The time for a round is the maximum of $K|S_i|$ and P. Now assume that $P > K|S_i|$. Then the time to encrypt is

$$t_e = rP \tag{20}$$

The time t_e will be smaller than that for the linear encryption of the message if

$$rP < K|M| \tag{21}$$

Therefore,

$$\begin{aligned} P &< \frac{K|M|}{r} \\ &= \frac{K|M|(c-1)}{i_{max}} \\ &= |S_i|K(c-1) \end{aligned} \tag{22}$$

This formula can be made more general by introducing the variable a, which indicates the number of cores used to implement the functions that comprise the overhead for parallel treatment of shards. This changes the value for P to be

$$P < |S_i|K(c-a) \tag{23}$$

2) The case of Variable-Sized Shards and Low Control Overhead: In this case, we consider a scenario where the PME has varying sizes, with the control overhead being smaller than the time required to encrypt a single shard. This involves shards of varying sizes, but restricts the overhead speed to the average size of the shards. Suppose the average size of a shard is represented as $|\overline{S_i}|$. In that case, the formula for reducing the overhead time follows the same analysis as in the section for shards of identical size. The time for this parallelization to encrypt the message is

$$t_{e,v} \leq rK|\overline{S_i}| \tag{24}$$

Different-sized shards mean that the concept of rounds becomes irrelevant. The varying sizes of shards allow for the shards to be interleaved and serviced as soon as they are finished. In this case, the overhead program remains operational for the entire time that the encryption action is in progress. Therefore, in general, The reduction is Δ

$$\Delta = \frac{|M|}{r|\overline{S_i}|} \tag{25}$$

3) The case of Variable-Sized Shards and High Control Overhead: In this case, we consider where the PME has varying sizes, with the control overhead being greater than the time required to encrypt a single shard. This occurs when the overhead time required to service the parallel loading and encryption exceeds the time spent on encryption itself. That is when,

$$P > K|\overline{S_i}| \tag{26}$$

or if P is larger then

$$\Delta = \frac{K|M|}{rP} \tag{27}$$

and

$$P < |\overline{S_i}|K(c-a) \tag{28}$$

The reduction is Δ

$$\Delta = \frac{K|M|}{rK|S_i|} = \frac{|M|}{r|S_i|} \tag{29}$$

or if P is larger then

$$\Delta = \frac{K|M|}{rP} \tag{30}$$

The larger the number of cores assigned to processing the encryption, the larger the value of $|S_i|$ and the larger the number of blocks processed in the same round, as well as potentially the smaller the number of rounds. This also means that it is more likely that P can be completed in a single core or thread.

IV. Conclusion and Future Work

In this paper, we demonstrated the increased encryption speed possible in polymorphic encryption. Modern computers are designed with additional processing cores, threading, and can utilize devices such as GPUs. Most of the newer "hard" or "secure" ciphers are block ciphers that utilize modes. Ciphers relying on feedback or feed-forward schemes, such as CBC, PCBC, OFB, or CFB, all require information from preceding encrypted blocks, requiring messages using them to be linearly encrypted. Parallelization is not possible for such ciphers. As long as the overhead function that schedules the cores can be implemented with a sufficient number of cores, so that at least two cores can be dedicated to the encryption process, polymorphic encryption will be faster. The more cores that can be devoted to encryption and decryption, the greater the time savings will be.

At this time (2025), this technique has not been applied to computers employing GPUs. Sufficient examples of using GPUs in other applications have already demonstrated that GPUs can be utilized as if they were computing cores, provided the application is primarily based on mathematical processing, such as encryption. An implementation using GPUs is necessary to demonstrate that the solution is both viable and faster.

With the characterization of speed increases possible with polymorphic ciphers, a more comprehensive treatment is also required that summarizes the increase in security. A study

should also characterize the optimal shard sizes for polymorphic encryption. Testing is necessary to quantify the effect, and the data gathered is essential for extended analysis of the speed increases related to the shard size.

Polymorphic RNGs are still a relatively new concept and also require additional study. Analysis in the subject requires characterization of the number of characters needed to break an RNG, denoted by the value ρ. Characteristics for each RNG/polyRNG should be recorded and made available to users, enabling them to utilize the RNGs effectively. A study of the effect of combining these RNGs is also needed.

REFERENCES

[1] Albert Carlson and Robert Le Blanc. Polymorphic encryption engine, 2015.

[2] Mario Stipčević and Çetin Kaya Koç. *True Random Number Generators*, pages 275–315. Springer International Publishing, Cham, 2014.

[3] Kamalika Bhattacharjee and Sukanta Das. A search for good pseudo-random number generators: Survey and empirical studies. *Computer Science Review*, 45:100471, 2022.

[4] Elena Almaraz Luengo. A brief and understandable guide to pseudo-random number generators and specific models for security. *Statistics Surveys*, 16(none):137 – 181, 2022.

[5] K Sathya, J Premalatha, and Vani Rajasekar. Investigation of strength and security of pseudo random number generators. *IOP Conference Series: Materials Science and Engineering*, 1055(1):012076, feb 2021.

[6] Rafael Álvarez, Francisco Martínez, and Antonio Zamora. Improving the statistical qualities of pseudo random number generators. *Symmetry*, 14(2), 2022.

[7] Conor Ryan, Meghana Kshirsagar, Gauri Vaidya, Andrew Cunningham, and R Sivaraman. Design of a cryptographically secure pseudo random number generator with grammatical evolution. *Scientific reports*, 12(1):1–10, 2022.

[8] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, Boston, MA, 1997.

[9] P. L'Ecuyer. Random numbers for simulation. *Communications of the ACM*, pages 85 – 98, 1990.

[10] Rod Downey and Denis R. Hirschfeldt. Algorithmic randomness. *Communications of the ACM*, 62(5):70 – 80, 2019.

[11] Haytham Idriss, Pablo Rojas, Sara Alahmadi andTarek Idriss, Albert Carlson, and Magdy Bayoumi. Shadow pufs: Generating temporal pufs withproperties isomorphic to delay-based apufs.

[12] Albert Carlson, Mandeep Singh, and Benjamin Williams. Basics of polymorphism in encryption. *2025 IEEE 6th Annual World AI IoT Congress, Seattle, WA*, 2025.

[13] Yelai Feng, Huaixi Wang, Chao Chang, Hongyi Lu, Fang Yang, and Chenyang Wang. A novel nonlinear pseudorandom sequence generator for the fractal function. *Fractal and Fractional*, 6:589, 10 2022.

[14] C. B. Roellgen. A generalized model for polymorphic ciphers, 2002.

[15] Albert Carlson, Indira K. Dutta, Bhaskar Ghosh, and Michael Totaro. Modeling polymorphic ciphers. *Sixth International Conference on Fog and Mobile Edge Computing (FMEC)*, 2021.

[16] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption, Fifth International Workshop Proceedings (March 1998)*, pages 168 – 188. Springer-Verlag.

[17] Claude Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656 – 715, 1949.

[18] John Earl Haynes and Harvey Klehr. *Venona: Decoding Soviet Espionage in the United States (Yale Nota Bene)*. Yale University Press, New Haven, CT, 1999.

[19] Jun Wu and Tei-Wei Kuo. Real-time scheduling of cpu-bound and i/o-bound processes. In *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No. PR00306)*, pages 303–310. IEEE, 1999.

[20] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel technology journal*, 6(1), 2002.

[21] Umesh Gajanan Nawathe, Mahmudul Hassan, King C Yen, Ashok Kumar, Aparna Ramachandran, and David Greenhill. Implementation of an 8-core, 64-thread, power-efficient sparc server on a chip. *IEEE Journal of Solid-State Circuits*, 43(1):6–20, 2008.

[22] Hitoshi Oi. Evaluation of ryzen 5 and core i7 processors with spec cpu 2017. In *2019 IEEE International Systems Conference (SysCon)*, pages 1–6. IEEE, 2019.

[23] Gerassimos Barlas. *Multicore and GPU Programming: An Integrated Approach*. Morgan Kaufmann, Burlington, MA, 2^{nd} edition.

[24] Wen mei W. Hwu, David B. Kirk, and Izzat El Hajjand. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, Cambridge, MA, 4th edition.

[25] Richard N Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19(1):57–84, 1983.

[26] Robert W. Sebesta. *Concepts of programming languages*. Addison-Wesley., Boston, MA, 9th edition.

[27] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons Inc., New York, 2nd edition, 1996.

[28] Shameem Akhter and Jason Roberts. *Multi-core programming*, volume 33. Intel press Hillsboro, Oregon, 2006.