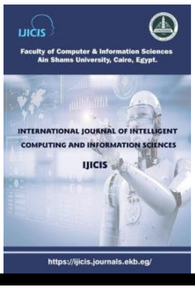
IJICIS, Vol.24, No.3, 83-92
DOI :10.21608/ijicis.2024.315432.1349

International Journal of Intelligent Computing and Information Sciences

https://ijicis.journals.ekb.eg/



ENHANCED AHO-CORASICK ALGORITHM FOR NETWORK INTRUSION DETECTION SYSTEMS

Anas Abbas*

Computer Systems Department,
Faculty of Computer and information sciences, Ain Shams University,
Cairo, Egypt
anas.mohammed@cis.asu.edu.eg

Mahmoud Fayez

Computer Systems Department,
Faculty of Computer and information sciences, Ain Shams University,
Cairo, Egypt
mahmoud.fayez@cis.asu.edu.eg

Heba Khaled

Computer Systems Department,
Faculty of Computer and information sciences, Ain Shams University,
Cairo, Egypt
heba.khaled@cis.asu.edu.eg

Said Ghoniemy

Computer Systems Department,
Faculty of Computer and information sciences, Ain Shams University,
Cairo, Egypt
ghoniemy1@cis.asu.edu.eg

Received 2024-08-25; Revised 2024-08-25; Accepted 2024-09-19

Abstract: *The exponential growth in internet data usage has created a pressing demand for highly efficient Network Intrusion Detection Systems (NIDS) capable of scaling with ever-increasing bandwidths to safeguard sensitive information. A cornerstone of NIDS, packet inspection, hinges on the ability to rapidly identify and analyze patterns within incoming data streams. The more diverse and extensive the pattern database, the more robust and effective the NIDS becomes. While the parallel failure-less version of the Aho-Corasick (AC) algorithm provides maximum parallelism, it faces significant memory constraints due to the large transition tables generated when dealing with a vast number of patterns. To mitigate this limitation and enhance the scalability of NIDS, we introduce a novel parallel failure-less compressed hashed variation of the Aho-Corasick algorithm. Our proposed approach leverages the power of compression and hashing techniques to significantly reduce memory consumption without compromising performance. Empirical evaluations demonstrate that our algorithm requires only a fraction (approximately the square root) of the memory footprint compared to the original parallel failure-less Aho-Corasick algorithm, making it a more practical and scalable solution for modern NIDS architectures.*

Keywords: *Compression, Hashing, NIDS, GPU, Parallel, AhoCorasick, Collissionless, failure-less, distributed system*

***Corresponding Author**: Anas Abbas

Computer Systems Department, Faculty of Computer and Information Science, Ain Shams University, Cairo, Egypt

Email address: anas.mohammed@cis.asu.edu.eg

1. Introduction and Motivation

The rapid escalation in internet usage and the involvement of the modern IoT technologies [1] has led to a significant increase in cyber threats and attacks [2]. Especially the rapid increase of usage of smart phones which lead to increased number of attacks [3]. Which some mobile app developer develops obfuscated applications [4] to protect against cyber-attacks as the attackers are now using a modern technique like moving target defense to deceive systems [5]. This increase in cyber threats poses severe risks to global economies [6], with substantial financial losses and disruptions to critical infrastructure becoming increasingly common. As businesses and governments grow more reliant on digital systems, the potential for devastating cyber-attacks has surged, driving a heightened demand for monitoring network traffic and identifying potential security breaches. Some systems even employ machine learning techniques to monitor the network for any potential threats.

Consequently, investments in cybersecurity systems have been intensified [7], with Network Intrusion Detection Systems (NIDS) emerging as critical components in modern cybersecurity. NIDS serve as the frontline defense in identifying and mitigating malicious activities within a network, relying heavily on signature-based detection [8], a technique that identifies known patterns of malicious activities. However, the packet inspection process, a key element of signature-based NIDS, is resource-intensive, particularly as the volume and variety of cyber threats expand, necessitating the ability to scale and process large datasets in real-time.

To address this challenge, parallel NIDS implementations utilizing Graphics Processing Unit (GPU) architecture have emerged as a powerful solution, leveraging GPUs' massively parallel processing capabilities to accelerate pattern matching processes and enhance threat detection and response.

Despite these advantages, modern GPU architectures are often constrained by memory bandwidth and capacity, limiting the performance of algorithms as pattern sets grow in size and complexity. This limitation is especially evident in the traditional parallel failure-less Aho-Corasick (AC) algorithm [9], which, despite maximizing parallelism, generates large transition tables that quickly consume available memory.

To overcome these challenges, this research proposes a novel approach: a parallel failure-less compressed hashed variation of the Aho-Corasick algorithm. By employing advanced compression and hashing techniques, this method reduces the memory footprint while maintaining high levels of parallelism, representing a significant advancement in the development of scalable and efficient NIDS, capable of protecting against the ever-growing threat of cyber-attacks in an increasingly connected world.

Our proposed variation of the Aho-Corasick algorithm is designed for optimal performance on distributed systems, ensuring work efficiency and overcoming memory limitations. This allows it to accommodate a larger number of patterns, making it more robust. In contrast, the original parallel Aho-Corasick algorithm struggles with work inefficiency due to significant overlap in tasks performed by different threads.

2. Related work

The problem addressed by related algorithms is formally known as "exact string matching," where the goal is to find all occurrences of a given pattern P (of length m) within a text T (of length n). Several algorithms have been developed to efficiently solve this problem, each offering distinct advantages and suited for specific use cases. However, only a few algorithms can be extended to support multiple patterns, and even fewer can run in parallel while achieving significant speedup. each of these algorithms has its own complexities and challenges e.g. the Boyer-Moore, Aho-Corasick, and Rabin-Karp algorithms.

2.1. Boyer-Moore

To enable the Boyer-Moore algorithm to support multiple pattern searches, it is essential to preprocess each pattern by constructing its "bad character heuristic array." This preprocessing allows the algorithm to skip unnecessary character comparisons, resulting in linear time complexity. While the parallel version of Boyer-Moore scales well [10], its scalability diminishes when patterns frequently occur in the text, as no character comparisons can be skipped. This can lead to a worst-case running time of $O(m \times n)$ when patterns are found throughout the input text.

2.2. Rabin-Karp

Rabin-Karp is one of the most effective algorithms for parallel execution and supports multiple pattern matching through various parallel variations [11]. Its performance improves significantly when using a collision-free hashing function, which eliminates the need for character-by-character comparisons [12]. However, a key limitation of all Rabin-Karp variations is the requirement that all patterns must be of the same length, which can be a significant drawback for some rule-based Network Intrusion Detection Systems (NIDS).

2.3. Aho-Corasick

Unlike Boyer-Moore and Rabin-Karp, Aho-Corasick inherently supports multiple patterns matching with patterns of different lengths. However, running Aho-Corasick in parallel on a GPU presents challenges due to the memory limitations of GPU architecture. The number of patterns it can support is constrained by the available GPU memory, and performance suffers significantly if this capacity is exceeded [13]. Some variations of the parallel Aho-Corasick algorithm remove the failure links [14], reducing memory usage and allowing more patterns to fit. However, these variations struggle to maintain efficient performance as the number of available threads increases.

3. Methodology

The proposed variation of the Compressed Hashed Parallel Failure-less Aho-Corasick algorithm utilizes collision-less hashing for pattern storage. Instead of storing each pattern in memory, we save only their hash values. This approach significantly reduces memory requirements, replacing a large tree with a vast number of nodes with a more compact structure. Each node in the tree stores only two numbers: the first is the hash value of the string represented by the graph edge, and the second is the length of that string. The pattern matching process is then performed using a regular serial graph traversal algorithm, such as BFS or DFS. During traversal, pruning techniques are applied, ensuring that at most N nodes are accessed during the matching process. On average, a couple of nodes are accessed in the case of a mismatch, and around log2(n) nodes are accessed in the case of a match, where n is the number of patterns. The tree construction process consists of three main steps: Trie building, Trie compression, and Trie collision-less hashing. Finally, parallel pattern matching is executed using a graph traversal algorithm.

3.1. Trie Building

In the Trie building step, words are inserted into a tree structure by adding nodes for each character, starting from a shared root. As words are added, shared prefixes follow common paths, reducing redundancy. Each node represents a character, with the end of a word marked at its final character's node. This structure organizes words efficiently, enabling quick retrieval and minimizing space usage for shared prefixes. For example, consider the patterns {"banana", "nabd", "bcdef", "bcfeg", "aaaaaa", "aabaa"}. Figure *1* shows the final state of the trie after all these patterns have been inserted. Notice that we can have a worst case of N * M nodes at worst case where N is the number of patterns and M be the length of each pattern. Usually, each node contains A pointers where A is the number of characters in the alphabet.

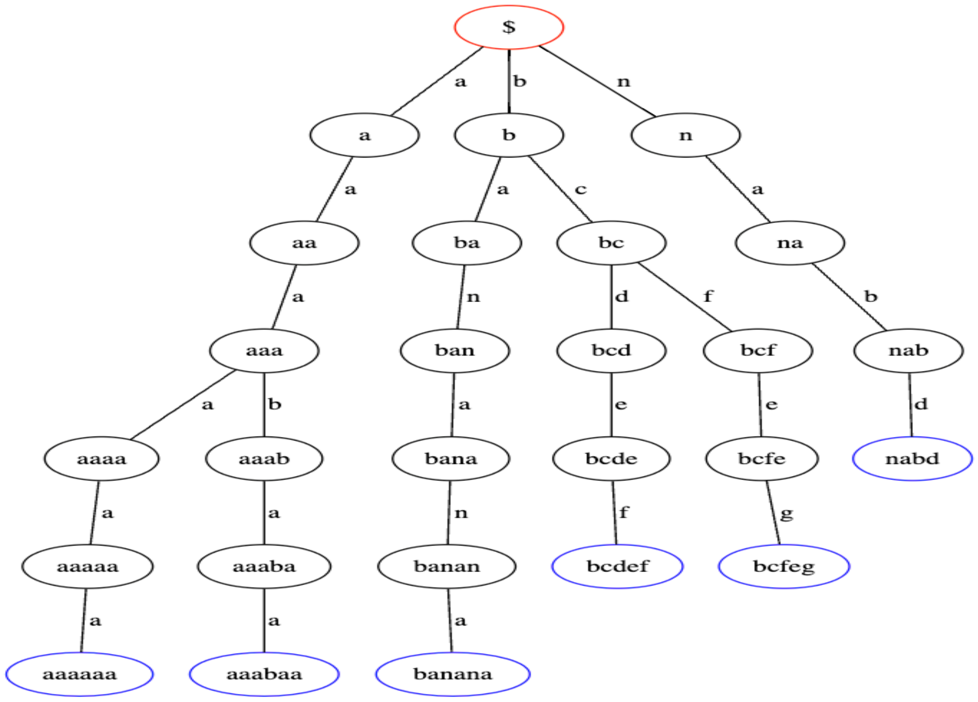


Figure 1 Regular Trie after Building

3.2. Trie Compression

In the Trie compression step, we reduce the size of a trie by merging nodes that have only one child. This process collapses linear chains of such nodes into a single node. The resulting structure, known as a "compact trie" or "radix tree," represents entire sequences of single-child nodes with just one node. This compression makes the trie more space-efficient while still allowing fast searches and retrieval of words or patterns. By reducing the overall memory usage, this technique also enhances the performance of certain operations, especially in tries that contain long, unique sequences. Considering the Trie built in Fig.1 we present the compressed version of the same trie after compression in Figure *2*.

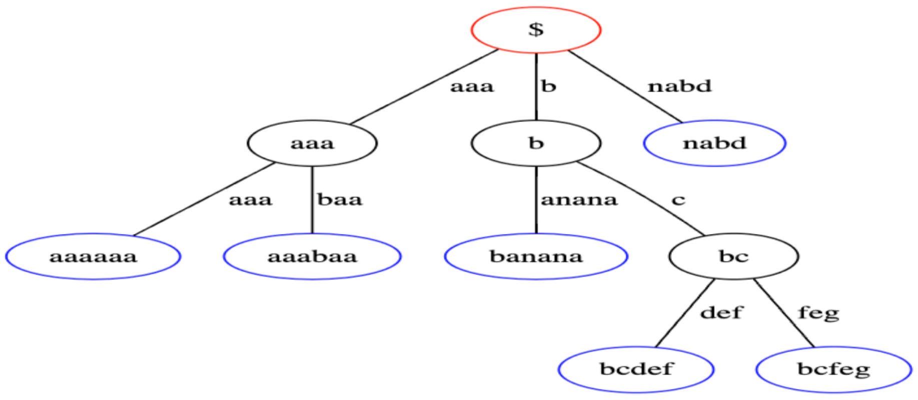


Figure 2 Built Trie after Compression

3.3. Trie Hashing

In the Trie hashing step, we replace long strings with two numbers and one Boolean flag the two numbers represents the hash value of the string and the length of the string whereas the Boolean flag determines whether or not the current node is a terminal node ie end of a string. We use a strong hashing function to minimize the risk of collisions, so we don't need to worry about them. This approach significantly reduces the size of each node. Considering the compressed trie built in Figure 2. *We present the same compressed trie after being hashed in*

Figure *3*. Note that string values in the figure are only for illustration purposes.

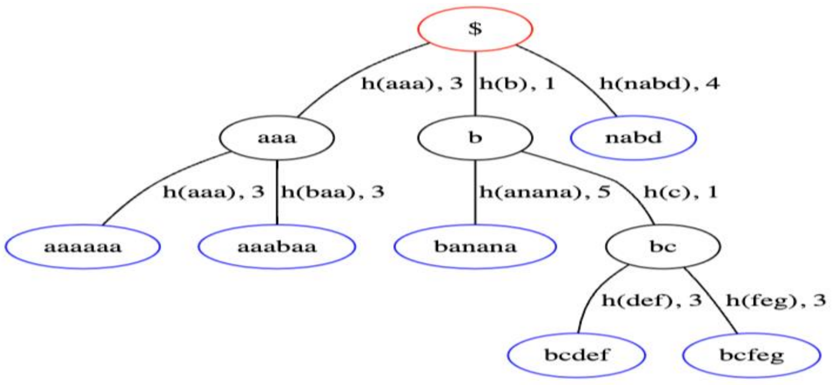


Figure 3 Built Trie after Compression and Hashing

3.4. Parallel Pattern Matching

In the parallel pattern matching step, we can begin traversing the graph from each index of the input text. For example, if we start at position i in the input text, we match against the same set of input patterns: {"banana", "nabd", "bcdef", "bcfeg", "aaaaaa", "aabaa"}. We use the compressed, hashed trie that we built earlier (as shown in Figure 3) to find the hashes for the substrings starting at position i and ending at positions i+1, i+3 and i+4. If any of those hashes matches, we continue traversing the graph recursively. In Table *1* Pseudo code for Parallel Trie Traversal, we present the pseudo code of the Depth First Search (DFS) traversal algorithm for the compressed hashed trie. The parallel DFS is preferred for the parallel execution rather than the BFS as the BFS algorithm introduces extra complexity for the shared queue between threads which can lead to lots of read/write conflicts [15].

Table 1 Pseudo code for Parallel Trie Traversal

Pseudo code

```
Function DFS(textStartIdx, textCurrentIdx, curNode):
    // Check if the current node is a terminal node (end of a pattern)
    If curNode.terminalNode is true:
        proposedPatternIdx = curNode.patternIdx
        proposedPatternHash = patternHashes[proposedPatternIdx]

        // Calculate the hash offset
        proposedPatternHashOffseted = (proposedPatternHash * lookupTable[textStartIdx]) % HASHVAL

        // Check if the proposed pattern's hash matches the text's hash in the interval
        If proposedPatternHashOffseted == intervalTextHash(textStartIdx, textCurrentIdx - 1):
            // If match found, add pattern index to matches
            Add proposedPatternIdx to matches

    // Traverse through all children of the current node
    For each child in curNode.children:
        edgeHash = child.edgeHash
        stringSize = child.stringSize

        // Calculate the target end index for the current substring
        targetEndIdx = textCurrentIdx + stringSize - 1

        // Get the hash of the corresponding substring in the text
        correspondingTextHash = intervalTextHash(textCurrentIdx, targetEndIdx)

        // Calculate the offset hash for the edge
        edgeHashOffseted = (edgeHash * lookupTable[textCurrentIdx]) % HASHVAL

        // If the corresponding text hash matches the edge's hash, continue DFS
        If correspondingTextHash == edgeHashOffseted:
            Call DFS(textStartIdx, targetEndIdx + 1, child.node)
```

3.4. Parallel Pattern Matching

- The steps of Trie Building, Trie Compression, and Trie Hashing can all be preprocessed before the input text is passed to the system. As a result, these steps do not impact the runtime during the actual text processing.
- Given that each non-leaf node has at least two children and there are at most N leaf nodes, the trie will have at most 2N nodes. Since it is a tree structure, there will be 2N - 1 edges, with each edge storing only two numbers and a flag.
- In the parallel matching step, we can execute the DFS function simultaneously at every position in the input text, as there are no data dependencies between these operations.
- For each character in the input text, on average, only a few nodes will be accessed in the case of a mismatch. In the case of a match, on average, $\log_2(N)$ nodes will be accessed, making the process highly efficient, meaning that overall time complexity would be $O\left(\frac{Text.\text{length} * \text{log2(N)}}{\tau}\right)$ where τ is the number of available threads.

4. Results

In our experiments, we considered various numbers of input patterns, each with different lengths. These patterns were matched against an input text of a constant 10^6 characters long (≈ 1Mbytes).
The input string and the patterns are randomly generated over the alphabet size $Alphabet = 26$ which is all English lower-case letters. During our experimentation, we ran both the parallel failure-less Aho-Corasick algorithm and the proposed compressed hashed failure-less Aho-Corasick algorithm. For each experiment, we measured the average running time per GPU thread for each character in the input text (in nanoseconds) and the number of nodes that needed to be copied to the GPU memory. Finally, we created two three-dimensional graphs to illustrate how the time and space complexity of both algorithms behave across all possible combinations of average pattern length and number of patterns.

4.1. Execution Time

We illustrate in Figure *4* how the execution times of both algorithms are affected. Our proposed algorithm maintains its performance, even as the number of patterns and the average pattern length increase, consistently running about five times slower than the regular parallel failure-less Aho-Corasick algorithm. This constant slowdown is acceptable because it is caused by the time required to hash the input text and match hashes, unlike the original algorithm, which relies on simple character-to-character matching. Hashing involves more complex operations, such as power and modulo, which contribute to the increased execution time.

4.2. Number of Nodes (Memory Usage)

Memory usages graph shown in Figure 5 illustrates the number of nodes each algorithm requires to be copied to the GPU memory for the pattern-matching process to start. Our proposed algorithm significantly reduces the number of nodes that need to be stored in GPU memory. The experimental results show that the memory space required by our proposed algorithm has a linear relationship with the overall size of the input patterns. In contrast, the parallel failureless Aho-Corasick algorithm exhibits a quadratic increase in the number of nodes needed, which depends on both the size of the input patterns and the number of alphabetic characters used in the system.

In the worst-case scenario, where there are 512 patterns with an average length of 512 characters each, the original Aho-Corasick algorithm requires approximately 250,000 nodes, while our proposed algorithm needs only around 500 nodes. Additionally, depending on the implementation details, the nodes in the compressed hashed parallel failureless Aho-Corasick algorithm often take up less memory than the original failureless Aho-Corasick nodes. This means that our approach not only significantly reduces the number of nodes but also decreases the memory required for each node.

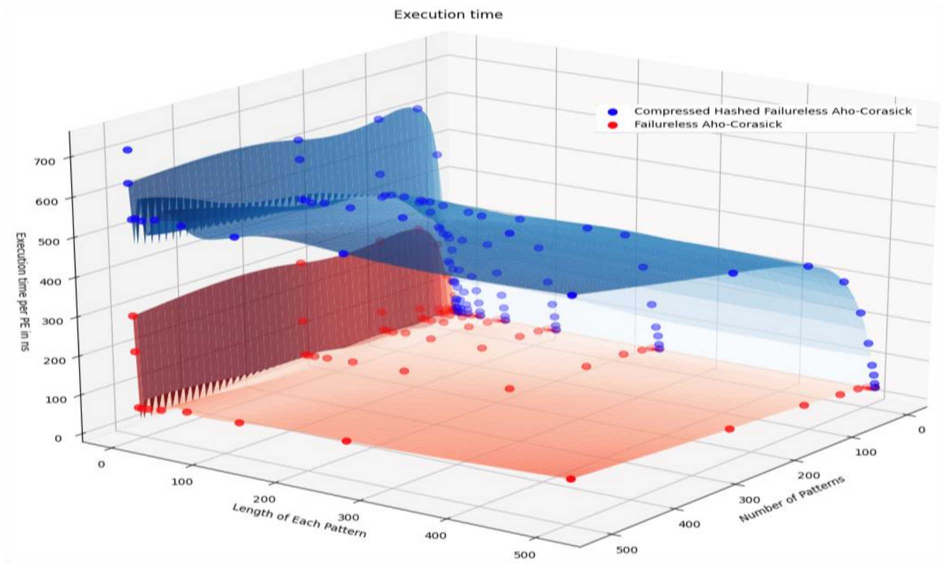


Figure 4 Execution Time Comparison

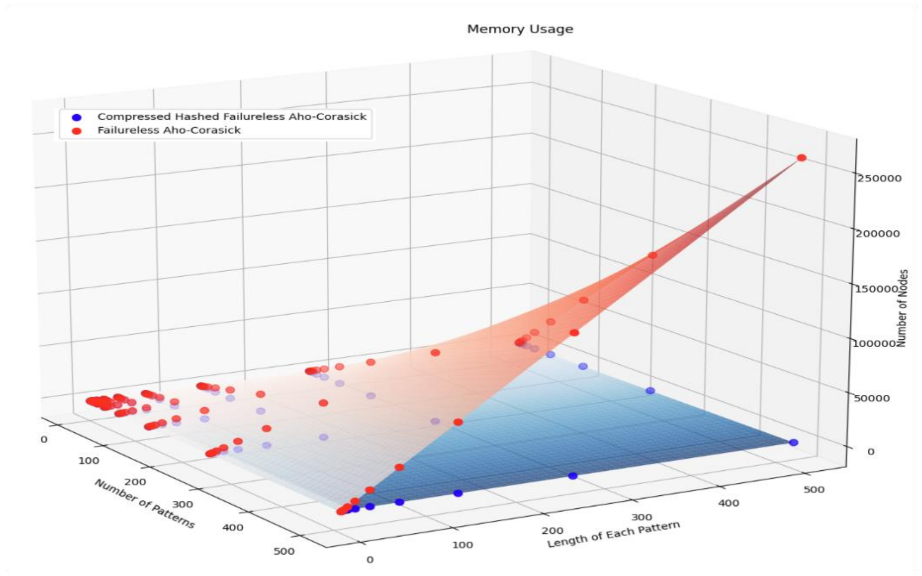


Figure 5 Memory Usages Comparison

4.2. Hardware Specs

Execution times were collected using the NVIDIA Nsight profiling system where an NVIDIA A100-PCI graphics card and an Intel(r) Xeon(r) silver 4314 CPU @ 2.40ghz were used to execute the two implementations.

5. Conclusion

Our proposed algorithm has demonstrated a significant reduction in memory usage, which is crucial for modern GPU architectures. Our experiments show that the compressed hashed version of the parallel failure-less Aho-Corasick algorithm uses only the square root of the memory required by the original version. This reduction in memory usage comes at the cost of a 4-5 times slower execution time due to the overhead of calculating hashes for the input text, which involves complex operations like exponentiation, modulo, and multiplication. Despite this trade-off, the proposed algorithm is highly beneficial for Network Intrusion Detection Systems (NIDS) with large pattern databases that are constrained by GPU memory limitations, as it effectively alleviates those memory constraints.

6. Author's Declarations and Statements

The study was supported by the Faculty of Computer & Information Science, Ain Shams University under the supervision of the computer systems department where we received general guidance and access to computing clusters.
We, the authors of this research declare that the research has no conflicts of interest and doesn’t include any work that involves human participants.

6.1. Competing interests

We, the authors of this research declare that no financial funding was received by any organization, and we have no financial or proprietary interests in any material discussed in this article.

6.2. Data sources

All of the input data used to test the performance of this research are randomly generated during the execution of our program and all parameters that were used to generate the data are well explained and shown in this article.

References

[1] A. Atta, “Controlling and Security System Using IOT Infrastructure and Image Processing in Educational Institutions,” *International Journal of Intelligent Computing and Information Sciences*, vol. 23, no. 4, pp. 128–141, Dec. 2023, doi: 10.21608/ijicis.2023.224742.1285.

[2] M. Mijwil, O. J. Unogwu, Y. Filali, I. Bala, and H. Al-Shahwani, "Exploring the Top Five Evolving Threats in Cybersecurity: An In-Depth Overview," *Mesopotamian Journal of Cyber Security*, pp. 57–63, Mar. 2023, doi: 10.58496/mjcs/2023/010.

[3] sara shehata, I. Hegazy, and E.-S. El-Horabty, "A STUDY FOR MALWARE STATIC ANALYSIS CLASSIFICATION ALGORITHMS WITH DIFFERENT FEATURES EXTRACTORS'," *International Journal of Intelligent Computing and Information Sciences*, vol. 23, no. 4, pp. 19–32, Dec. 2023, doi: 10.21608/ijicis.2023.242171.1300.

[4] H. Aboud, H. Hindy, S. Gaber, and A.-B. Salem, "Artificial Intelligence based Algorithm for Detecting Android Obfuscated Applications," *International Journal of Intelligent Computing and Information Sciences*, vol. 0, no. 0, pp. 0–0, Mar. 2024, doi: 10.21608/ijicis.2024.250295.1308.

[5] A. Aly, M. Fayez, M. Al-Qutt, and A. Hamad, "Navigating the Deception Stack: In-Depth Analysis and Application of Comprehensive Cyber Defense Solutions," *International Journal of Intelligent Computing and Information Sciences*, vol. 23, no. 4, pp. 50–65, Dec. 2023, doi: 10.21608/ijicis.2023.247380.1306.

[6] O. Gulyás and G. Kiss, "Impact of cyber-attacks on the financial institutions," *Procedia Comput Sci*, vol. 219, pp. 84–90, 2023, doi: 10.1016/j.procs.2023.01.267.

[7] I. Fernandez De Arroyabe, C. F. A. Arranz, M. F. Arroyabe, and J. C. Fernandez de Arroyabe, "Cybersecurity capabilities and cyber-attacks as drivers of investment in cybersecurity systems: A UK survey for 2018 and 2019," *Comput Secur*, vol. 124, Jan. 2023, doi: 10.1016/j.cose.2022.102954.

[8] k. Azarudeen, S. H. Kumar, T. V Aswin Vijay, P. Thirukumaran, and V. S. B. Balaji, "Intrusion Detection System based on Pattern Recognition using CNN," in *2023 International Conference on Sustainable Computing and Smart Systems (ICSCSS)*, 2023, pp. 567–574. doi: 10.1109/ICSCSS57650.2023.10169670.

[9] A. V Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," 1975.

[10] A. Hnaif, K. Jaber, M. Alia, and M. Daghbosheh, "Parallel scalable approximate matching algorithm for network intrusion detection systems," *International Arab Journal of Information Technology*, vol. 18, no. 1, pp. 77–84, Jan. 2021, doi: 10.34028/iajit/18/1/9.

[11] L. S. N. Nunes, J. L. Bordim, Y. Ito, and K. Nakano, "A rabin-karp implementation for handling multiple pattern-matching on the gpu," *IEICE Trans Inf Syst*, vol. E103D, no. 12, pp. 2412–2420, Dec. 2020, doi: 10.1587/TRANSINF.2020PAP0002.

[12] A. Abbas, M. Fayez, and H. Khaled, "Multi-Pattern GPU Accelerated Collision-Less Rabin-Karp for NIDS," *International Journal of Distributed Systems and Technologies*, vol. 15, no. 1, 2024, doi: 10.4018/IJDST.341269.

[13] M. Najam-ul-Islam, F. T. Zahra, A. R. Jafri, R. Shah, M. ul Hassan, and M. Rashid, "Auto implementation of parallel hardware architecture for Aho-Corasick algorithm," *Design Automation for Embedded Systems*, vol. 26, no. 1, pp. 29–53, Mar. 2022, doi: 10.1007/s10617-021-09257-7.

[14] H. J. Hadi, K. Shahzad, N. Ahmed, Y. Cao, and Y. Javed, "A Scalable Pattern Matching Implementation on Hardware using Data Level Parallelism," in *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2023, pp. 2530–2537. doi: 10.1109/TrustCom60117.2023.00354.

[15] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2010, pp. 303–314. doi: 10.1145/1810479.1810534.