International Journal of Distributed Systems and Technologies
Volume 15 • Issue 1

Multi-Pattern GPU Accelerated Collision-Less Rabin-Karp for NIDS

Anas Abbas, Ain Shams University, Egypt*
https://orcid.org/0009-0001-6583-6621
Mahmoud Fayez, Ain Shams University, Egypt
Heba Khaled, Ain Shams University, Egypt

ABSTRACT

In the domain of network communication, network intrusion detection systems (NIDS) play a crucial role in maintaining security by identifying potential threats. NIDS relies on packet inspection, often using rule-based databases to scan for malicious patterns. However, the expanding scale of internet connections hampers the rate of packet inspection. To address this, some systems employ GPU accelerated pattern matching algorithms. Yet, this approach is susceptible to denial of service (DOS) attacks, inducing hashing collisions and slowing inspection. This research introduces a GPU-optimized variation of the Rabin-Karp algorithm, achieving scalability on GPUs while resisting DOS attacks. Our open-source solution (https://github.com/AnasAbbas1/NIDS) combines six polynomial hashing functions, eliminating the need for false-positive validation. This leads to a substantial improvement in inspection speed and accuracy. The proposed system ensures minimal packet misclassification rates, solidifying its role as a robust tool for real-time network security.

KEYWORDS

Aho-Corasick, Rabin karp, collisionless, CUDA, CUDACUB, DDOS attack, GPU, NIDS, Pattern matching, hashing

INTRODUCTION AND MOTIVATION

The proliferation of telecommunication infrastructures, along with affordable computing devices and smartphones, has spearheaded a significant expansion of the Internet, leading to a substantial increase in internet bandwidth and users over the past few years. With this digital evolution, the internet has increasingly become a cornerstone of our daily activities, making it a lucrative target for cybercriminals.

This unprecedented growth of internet usage in recent years has triggered a corresponding surge in cyber threats and attacks (Mijwil et al., 2023). The rise in internet usage is well-documented, from approximately 413 million users at the start of the twenty-first century to almost 4.7 billion users by 2022, signifying that nearly 60% of the world's population actively use the internet (Li &

DOI: 10.4018/IJDST.341269 *Corresponding Author

Liu, 2021), Also in 2021, Positive technologies specialists (*Cybersecurity Threatscape: Year 2021 in Review Positive Technologies*, n.d.) recorded more than 2,400 attacks, that is 6.5 percent more than 2020 attacks stating that this huge increase is due to the corona virus and the heavy use of the internet and working from home. This expansion in internet usage has inevitably increased the number of potential targets for cybercriminals, leading to a sharp rise in cyberattacks as it has a huge impact on the economy (Gulyás & Kiss, 2023) and as a result, cyber-attacks drive investment in cybersecurity systems (Fernandez De Arroyabe et al., 2023), all of that proves that a NIDS is crucial for any device.

The critical role of Network Intrusion Detection Systems (NIDS) cannot be overemphasized in today's hyperconnected world as shown in the risk analysis research published in the International Journal of Advanced Computer Science and Applications (Jakim et al., n.d.). As technological advancements continue to expand, so does the complexity and size of network infrastructure, creating more opportunities for attackers to exploit vulnerabilities as stated in the neural Computing and Applications survey (Keserwani et al., 2023). The variety and increasing frequency of cyber-attacks necessitate reliable and robust NIDS that can efficiently mitigate these security risks.

Recent studies illustrate the vital part played by NIDS in ensuring network security. A systematic study published in the European Transactions on Telecommunications (Ahmad et al., 2021) highlighted how NIDS scrutinizes network traffic to ensure its confidentiality, integrity, and availability, effectively thwarting potential intrusions. In the realm of intrusion detection, two fundamental strategies are employed: Anomaly-based IDS and Signature-based IDS (Liao et al., 2013). Signature-based IDS focuses on identifying intrusion occurrences through pre-defined "signature" patterns of known attacks. To remain effective, it regularly updates its signature database to detect the latest trends and zero-day attack patterns where the quality of those signatures influences the overall effectiveness of the NIDS as stated by a study published in the Information Security Journal (Sommestad et al., 2022) that demonstrated the popular Snort signature-based solution which proved to be a very effective ruleset as its latest release of Snort 3 which showed better performance than the earlier version (Boukebous et al., 2023). Conversely, the anomaly-based intrusion detection system (also known as Behavior-based Detection) works by comparing normal behavioral patterns with new activities, continuously monitoring network activities to flag potential intrusions.

Creating a successful NIDS remains a crucial challenge in the realm of network security. Although significant progress has been made in NIDS, the predominant focus lies on signature-based methods, largely neglecting anomaly detection techniques. Several factors contribute to the reluctance to embrace anomaly detection, such as the intricate behavioral dynamics of systems, the need for reliable training data collection, high associated costs, and error rates arising from the dynamic nature of the data whereas the signature-based NIDS proved to be resilient against Distributed Denial of Service Attack (DDOS) (Chen & Lai, 2023), (Sardar et al., n.d.).

For a signature-based NIDS, Pattern matching has been confirmed to be the most time-consuming process due to its critical role in identifying potentially harmful activities as it's a major functionality even in AI-based NIDS (Azarudeen et al., 2023; Siva Kumar et al., n.d.). The increasing transmission rate of packets, especially in high-traffic networks, places a significant load on pattern-matching algorithms. It requires these algorithms to be both efficient and swift to maintain pace with the network speed and ensure real-time processing; otherwise, the time complexity of pattern matching algorithms can lead to a slowdown in processing, which could result in packet drops, degraded system performance, and eventually security vulnerabilities as stated in the study published by A. Waleed, A. F. Jamali, and A. Masood [6].

The Rabin-Karp algorithm, being data-driven with minimal task and data dependencies, has demonstrated exceptional parallel scalability, encountering few bottlenecks necessitating sequential executions. To harness this scalability, we opted for a hybrid CPU/GPU architecture, which excels in handling a growing number of concurrent simple tasks within the Rabin-Karp algorithm. Unlike CPU-only architecture, which typically features only a limitation of the number

of cores that are in the double digits and are designed primarily for complex instructions, CPU/GPU hybrid architecture exhibits superior scalability for linearly increasing concurrent tasks in the Rabin-Karp algorithm. Additionally, this architecture minimizes memory latency by consolidating all data in shared and global memory, contrasting with CPU only which incurs latency when loading/unloading data. This strategic choice enhances the algorithm's overall performance on parallel architectures as the recent article published by (Groth et al., 2023) in the knowledge and information systems journal has proved to be 40% higher throughput than the CPU-only approach when dealing with string manipulation operations. Our approach offers a considerable speedup, especially when dealing with a large data set of patterns as mentioned in the 2023 study published Application of soft computing (Baloi et al., 2023) which used GPU-accelerated pattern matching to find similarity metrics. Two algorithms have emerged as the most widely utilized for multiple string matching on GPUs - the Aho-Corasick (AC) and Rabin-Karp algorithms.

The AC algorithm is undeniably efficient, yet its application is limited due to the inherent memory latency of GPUs. The algorithm's quick access to data to determine matches and the GPU's inability to store the substantial automaton required by AC in its shared memory create a challenge. As a result, the automaton must be stored on slower global memory (Çelebi & Yavanoğlu, 2023), reducing efficiency. As found by Najam-ul Islam (Najam-ul-Islam et al., 2022) in their experiments when the number of rules was increased to 10,000 the throughput declined, and more resources were needed to keep up with the increasing number of rules.

On the contrary, because the Rabin-Karp algorithm converts strings into hashes it bypasses this issue by requiring only a single memory call to compare if the current hash of a substring exists within precalculated pattern hashes stored in the GPU shared memory. Nevertheless, Rabin-Karp encounters difficulties due to hash collisions. Even if a current hash of a substring is found within the precalculated pattern hashes, it does not definitively indicate an actual match, necessitating a character-by-character validation to avoid false-positive matches.

The Rabin-Karp algorithm has seen significant advancements in modern implementations, where the utilization of reliable hash functions has effectively reduced the need for character-by-character validation. Despite these improvements, concerns about performance stability persist. Particularly, true matches can significantly impact the efficiency of the Rabin-Karp algorithm, leading to potential issues such as packet drop or reduced packet transmission rates. Consequently, it becomes crucial to address these performance-related challenges to ensure the algorithm's optimal functionality.

In this paper, we propose an innovative variant of the Rabin-Karp algorithm that eliminates the need for character-by-character validation. This novel approach yields a consistently high performance of the string-matching algorithm, which subsequently contributes to a more dependable Network Intrusion Detection System (NIDS). Notably, this method paves the way for future improvements in pattern-matching processes in high-traffic networks.

In this paper, we propose our Rabin-Karp algorithm implementation which is designed for efficient real-time data processing, strategically enhancing the traditional approach. A crucial improvement involves integrating a skip mechanism for character-by-character string validation, significantly boosting performance, particularly in real-time applications. This character validation skip expedites the algorithm's execution, making it ideal for processing substantial data volumes in real-time scenarios. The decision to select Rabin-Karp over Aho-Corasick and Boyer-Moore is driven by specific considerations:

1. **Memory Efficiency and Scalability:**
    - Rabin-Karp excels in supporting multiple pattern matching with limited memory, making it ideal for real-time intrusion detection.
    - The algorithm's scalable parallel approach ensures optimal utilization of computational resources.

2. **Boyer-Moore Pattern Occurrences Limitations:**
   - Boyer-Moore experiences performance degradation with frequent patterns in the input text, exposing it to intentional slowdowns or successful DDoS attacks.
3. **Aho-Corasick Memory Constraints:**
   - While Aho-Corasick scales well with increased computational resources, it is constrained by the need for a Deterministic Finite Automaton (DFA) to inspect the input text.
   - As the number of patterns grows, the DFA size significantly increases, limiting performance, especially in GPU architectures.

In summary, the Rabin-Karp algorithm stands out as the optimal choice for our real-time intrusion detection systems, striking a balance between memory efficiency, scalability, and resilience against potential attack scenarios.

RELATED WORK

The problem of string-matching is delineated as: given a Pattern array of size k where each $Pattern_i$ has a length of m and an input text x of length n, locate all instances of any $Pattern_i$ in the text x. The elementary algorithm to tackle this problem would involve contrasting every character of the input text with every character in every pattern, thus culminating in a considerable $O\left(k * m * n\right)$ total complexity.

Alternative string-matching algorithms like Knuth-Morris-Pratt (KMP) (Knuth et al., 1977), Rabin-Karp (Karp & Rabin, 1987), and Boyer-Moore (Boyer & Moore, 1977) are equipped to address this problem. Upon implementing certain strategies, these algorithms can be manipulated to work with multiple patterns. For instance, when applied to multiple patterns, KMP metamorphoses into the Aho-Corasick (Aho & Corasick, 1975) algorithm, which is essentially a more generalized form of KMP.

To enable the Boyer-Moore algorithm to support multiple pattern searches, it is crucial to preprocess each pattern to construct its "bad character heuristic array". This would aid in skipping unnecessary character comparisons, thereby resulting in linear time complexity. While the parallel version of Boyer-Moore scales well (Hnaif et al., 2021) the algorithm's scalability takes a hit when the patterns frequently occur in the text as no character comparisons can be skipped. This can lead to a worst-case $O\left(m * n\right)$ running time (S.DAWOOD, 2020) when patterns are found throughout the input text.

Unlike Boyer-Moore, Aho-Corasick does not have the same shortcomings. It preprocesses the given pattern to build a single Deterministic Finite Automaton (DFA) which is then traversed using the input text. During this traversal, if a transition to a terminal state transpires, a pattern match is identified, or it continues its standard traversal over the DFA. Owing to its demonstrated efficacy, Aho-Corasick is extensively utilized for pattern matching in most Network Intrusion Detection Systems (NIDS). Nonetheless, its efficiency diminishes in the context of graphical processing unit (GPU) parallel architecture due to data dependency issues. To execute Aho-Corasick in parallel, every block of threads requires a copy of the DFA in their shared memory. Alternatively, the DFA can be copied into global memory for every thread to access. Regrettably, both approaches do not scale well given the limitations of shared memory and the impracticality of frequent global memory access by all threads. The recent studies (Najam-ul-Islam et al., 2022; Papadogiannaki et al., n.d.) show how the GPU-accelerated version of the Aho-Corasick algorithm is significantly affected by the number of rules in any network intrusion detection system meaning that with the increasing number of rules, the system will most probably suffer from packet dropping when parallel AC is used.

The standard implementation of the Serial Rabin-Karp (SRK) algorithm generally performs well with an average and best-case running time of O(n + m). However, it exhibits a worst-case time complexity of O(n * m). This worst-case scenario materializes when all characters in both the pattern and the text have hash values matching those of all substrings of the text with a length equal to the pattern's length (n). A study published in the Journal of Science and Technology Research (Aigbe & Nwelih, 2021) provides insight into this behavior. The same is true for the parallel Rabin-Karp implementations (PRK) but we see a significant improvement in the execution times when using the full power of the GPU as the number of cores increases.

Table 1 and Table 2 illustrate the execution time in milliseconds of our implementations for the serial and parallel Rabin-Karp algorithm under normal conditions. In contrast, Table 6 displays execution times in milliseconds in a Denial of Service (DOS) attack condition, where the algorithm approaches its worst-case performance.

In this paper, we propose that the Rabin-Karp algorithm is optimally suited for GPU acceleration for use in NIDS when dealing with multiple pattern matching. The Rabin-Karp algorithm preprocesses every pattern into a single hash value and calculates hash values for every prefix of the input text. The matching process involves comparing hash values for each substring of the input string to pattern hash values. In case two hash values match, a character-by-character comparison is then conducted for the substring and the pattern with the identical hash value. Although Rabin-Karp may seem less efficient at first glance when compared to the Aho-Corasick with the character-by-character comparison eliminated in Rabin-Karp, our approach will outperform any other string-matching algorithms.

The idea of the elimination of character-by-character comparison to gain significant speedup in the running time was already in use in the study published by Abdullah Ammar and Hasan Bulut (Karcioglu & Bulut, 2021) where they improved their hashing technique to skip the character-by-character comparison for the last few characters (q characters) whereas in our research we eliminate the need for character-by-character for all the characters of the pattern and the text.

Table 1. Serial Rabin-Karp execution times under normal conditions

Number of patterns (k)	Length of each input pattern (m)		
	10	20	30
2^0	2690	3210	3160
2^2	2750	3210	3200
2^4	2950	3230	3220
2^6	3650	3210	3250
2^8	6280	3250	3260

Table 2. Parallel Rabin-Karp execution times under normal conditions

Number of patterns (k)	Length of each input pattern (m)		
	10	20	30
2^0	3.52	4.58	4.57
2^2	3.89	4.58	4.57
2^4	4.78	4.59	4.59
2^6	6.11	4.66	4.61
2^8	6.32	4.71	4.70

METHODOLOGY

We achieved high-speed multiple patterns matching in the Rabin-Karp algorithm by exploiting implementation speedups, parallel GPU architecture, and finally skipping the string validation part of the algorithm.

String validation is necessary for every hash-based pattern-matching algorithm as the hashing collisions produce false positive matches. The next section discusses in detail why it's safe to skip string validation. The universal hash function proposed by Dietzfelbinger et al. (Dietzfelbinger et al., n.d.), is defined as $h_a\left(x_0 \ldots x_{n-1}\right) = \sum_{i=0}^{n} x_i a^{n-i-1} mod\ p$, where x is the input string of length n, p is a Mersenne prime number and a is a randomly generated number such that $\left|u\right| \leq a < p$, u is the size of alphabet characters.

For architecture limitations, instead of just one hash function, we will be creating **six** different hashing function instances of the above hash function each one with a randomly generated seed a and a carefully selected Mersenne prime number p, finally, we do multiple hashing by concatenating the six hash values into one hash value.

Implementation Speedups

Our speedups are heavily based on the instruction set architecture for GPU and CPU where we carefully select the six Mersenne prime numbers to be $\left[524287, 131071, 8191, 127, 31, 7\right]$ so that firstly, we force the concatenation of the resultant concatenation of the six hashing functions to fit into 64-bit integer data type using bit masks as shown in Fig. 1, which saves lots of time when moving data around. Secondly, we replace the mathematical operation modulus with a combination of binary, plus, and branching operations which saves lots of CPU cycles.

The final formula of the hashing function is shown in Eq. (1) where all of the six hash functions are combined see Figure 1.

$$h_f\left(x\right) = h_{a0}\left(x\right).2^0 + h_{a1}\left(x\right).2^3 + h_{a2}\left(x\right).2^8 + h_{a3}\left(x\right).2^{15} + h_{a4}\left(x\right).2^{28} + h_{a5}\left(x\right).2^{45} \quad (1)$$

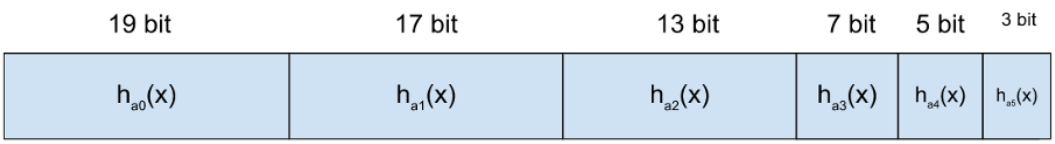
GPU Architecture

For this specific problem, we decided to use the CUDA Unbounded library (CUB) (Merrill, n.d.) to utilize its high-performance kernel components that can be tuned for this kind of implementation.

Skipping String Validation

The equality of two hash values doesn't necessarily assure that the two strings are the same. So, when we skip string validation, we come across two vulnerabilities. firstly, the system might encounter a false match occurring by a hashing collision. Secondly, the attacker can make use of the first vulnerability and try to do a false positive attack on our system.

Figure 1. Hashing combination



Hashing Collision Analysis

From Thorup's work (Thorup, 2015), we know that collision probability $P_{collision}$ for the above hashing function lies in the range of $\frac{1}{p} \leq P_{collision} \leq \frac{d}{p}$, where d is the length of the string to be hashed and p is the carefully selected Mersenne prime number, assuming the average case $P_{collision} = \frac{d}{2.p}$, now for a collision to happen, all of the six hash functions have to collide at the same time. From Bayes' theorem, we conclude that the probability of six hash functions' values colliding at the same time is $P_{6-collisions} = \frac{d^6}{2^6.\prod_{i=0}^{6} p_i}$ which means that if we assume our input text of size n we can calculate exactly how many times our hashing functions are going to be called using the formula $n-d+1$. when putting it all together we conclude that for each time our program with input size n the probability for at least one mismatch $P_{mismatch}$ is $\frac{(n-d+1).d^6}{2^6.\prod_{i=0}^{6} p_i}$.

To show how low the probability of P_{mismatch} is, let's assume that an average-sized packet is of size $n = 1500\,bytes$ and an average pattern length of $d = 20$ characters long pattern. When plugging those numbers in the above equation, we find out that $P_{mismatch}$ is pretty low and it is expected to process more than a **billion** packages to encounter a single mismatch, and even if a false positive is encountered, it would not impact the decision regarding whether a packet is malicious or not. Most systems require the presence of at least three false positives to affect the overall decision. To have three false positives in consecutive packets, it would require a massive amount of data, specifically billions multiplied by the average packet size of 50 bytes, resulting in a total number of bytes exceeding a billion zettabytes. It is worth noting that the 2021 Statista study (Statista Research Department, 2021) predicts the size of the internet to reach approximately 180 zettabytes by 2025. Given these considerations, it can be concluded that the occurrence of such an incident is highly unlikely, and it is safe to disregard the validation of false positives.

False Positive Attack

Let's assume a scenario where an attacker who has a copy of the patterns that our system is looking for and decided to do a DOS attack for our system by enforcing our system runs lots of string validations which results in a false match, he first has to correctly guess all of the seeds $[a_0, \ldots, a_5]$ for all of the six-hashing function instances that we used initially to generate our hash values where the probability of that $P_{seeds} = \frac{1}{\prod_{i=0}^{6} p_i}$, furthermore, he also needs to generate a random string that does a hashing collision and the probability of that is calculated above $P_{6-collisions}$.

When putting them together, we conclude that the effort needed for this kind of attack needs an astronomical number of operations $P_{seeds} * P_{6-collisions}$ when calculated we find it's a 36-digit number.

ALGORITHM

Our implementation for Multi-pattern GPU accelerated collision-less Rabin-Karp is divided into five main steps, Given the Multi-pattern *Pattern* array of size k where each pattern' length is m and the input text x of length n below is the high-level explanation of the five steps.

1. Compute and save the lookup table for all $a_i^j \bmod p_i$ such that $0 \le i \le 5$ and $0 \le j < p_i$, where p_i is the corresponding Mersenne prime number.
2. Compute hashes for all patterns in parallel $h_f\left(Pattern_i\right)$ where $0 \le i \le k$ and store it in a concurrent hash table HT such that $h_f\left(Pattern_i\right)$ is the key and i is the value.
3. Compute array S of length n using the input text x in parallel such that $S_i = (x_i.a_0^{n-i-1} \bmod p_0).2^0 + (x_i.a_1^{n-i-1} \bmod p_1).2^3 + \ldots + (x_i.a_5^{n-i-1} \bmod p_5).2^{45}$.
4. Compute prefix-sum array $\hat{S}$ such that $\hat{S}_j = \sum_{i=0}^{j} S_i$ for all j such that $0 \le j < n$, this step is done using "*decoupled look-back*" algorithm implemented in CUB Library (Merrill & Garland, n.d.).
5. For all j $\left(0 \le j \le n-m\right)$, compute $W = \left(\hat{S}_{j+m-1} - \hat{S}_{j-1}\right).a^{m-n-j}$, which is equal to the hash value of the substring $x_j, x_{j+1}, \ldots, x_{j+m-1}$, mark a match with pattern index $HT\left[W\right]$ if W in HT, skip otherwise.

For an input text "helloworld" and an input pattern "low" Figure 2 illustrates steps 3, 4, and 5. Assuming that we are using h_f as our hashing function.

Table 3 presents the pseudo-code of the implementation of the parallel Rabin-Karp algorithm for multiple patterns.

The key difference between (Nunes et al., 2020) implementation and our proposed implementation is that our proposed one uses our hashing function, parallel architecture, and finally, we skip the string validation phase.

Time Complexity Analysis

- In step1, can be done efficiently by looping from zero to the maximum value of all Mersenne prime numbers which is p_5 where in each iteration we compute $a_i^j \bmod p_i$ for each i. This results in a total complexity of $O\left(p_5 * h\right)$, where h is the number of hashes, and in that case it's $h = 6, p_5 = 2^{19} - 1$ leading to approximately **3 million** operations in total which is always constant no matter how big the input is, so we can just discard this step from the analysis.

Figure 2. Steps 3, 4, and 5 illustration

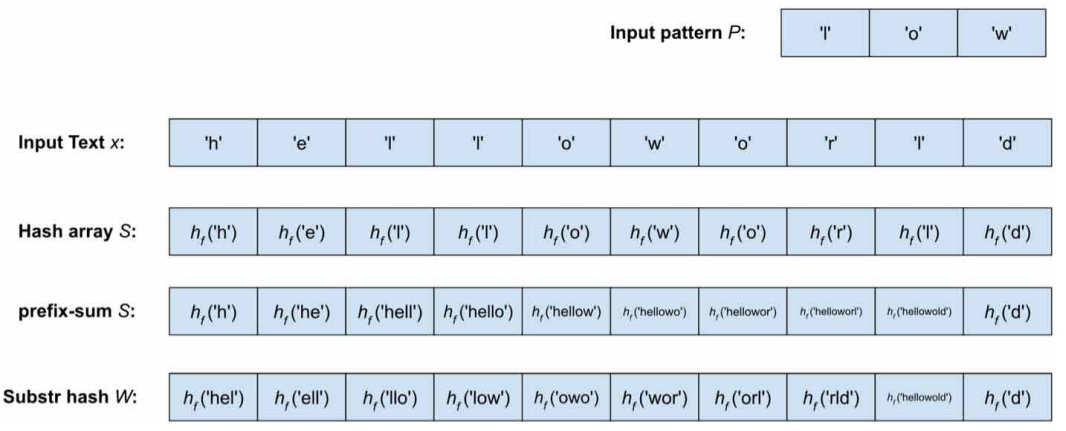


Table 3. Parallel Rabin-Karp pseudo code

Pseudo code	
Input:	
$Pattern$: k string array where each string is m $charachters\,long$ representing input patterns.	
$Text$: $n\,charachters\,long\,string$ representing input text.	
$\lvert u \rvert$: an integer number representing the size of the alphabet.	
random(s, e): a random generator function that returns a random integer in the interval [s, e].	
Output:	
$Match$: $n\,integers\,array\,where$ each i -th integer $x\,either$ -1 or fulfills the equation $Text_{i,i+1,\ldots,i+m-1} = Pattern_{x,x+1,\ldots,x+m-1}$	
1:	function Step1($P[6], S[6], A[6], L[P_5]$)
2:	for $i=0$ to P_5 do
3:	for $j=0$ to 6 do
4:	$L_i = L_i + \left(A_j^i \bmod P_j\right).2^{s_j}$
5:	end for
6:	end for
7:	end function
8:	function Step2($P[6], S[6], A[6], PHT[2^{64}, 2^{32}], Pattern[k][m]$)
9:	$ph=0$
10:	$pat = Pattern_{\gamma}$
11:	for $i=0$ to m do
12:	for $j=0$ to 6 do
13:	$ph = ph + \left(\left(\left(ph / 2^{s_j}\right) \wedge P_j\right).A_j + pat_i\right) \bmod P_j\Big).2^{s_j}$
14:	end for
15:	end for
16:	$PHT[ph] = \gamma$

continued on following page

Table 3. Continued

Pseudo code	
17:	end function
18:	function Step3($P\left[6\right], S\left[6\right], L\left[P_5\right], TH\left[n\right], Text\left[n\right]$)
19:	for $i=0$ to 6 do
20:	$TH_{\gamma} = TH_{\gamma} + ((((L_{(n-\gamma-1) \bmod (P_i-1)} / 2^{s_i}) \wedge P_i).Text_{\gamma}) \bmod P_i).2^{s_i}$
21:	end for
22:	end function
23:	function Step4 ($TH\left[n\right]$)
24: 25:	$TH_{\gamma} = \sum_{i=0}^{\gamma} TH_i$
26:	end function
27:	function Step5 $\left(P\left[6\right], S\left[6\right], PHT\left[2^{64}, 2^{32}\right], TH\left[n\right], Match\left[n\right]\right)$
28:	if $\gamma + m - 1 < n$ then
29:	$hash = 0$
30:	for $i=0$ to 6 do
31:	$tmp = \left(TH_{\gamma+m-1} / 2^{s_i}\right) \wedge P_i$
32:	if $\gamma > 0$ then
33:	$tmp = tmp - \left(TH_{\gamma-1} / 2^{s_i}\right) \wedge P_i$
34:	end if
35:	$tmp = (tmp \ . \ ((L_{m-n+j} / 2^{s_i}) \wedge P_i)) \bmod P_i$
36:	$hash = hash + (tmp \ .2^{s_i})$
37:	end for
38:	$Match_{\gamma} = PHT\left[hash\right]$
39:	end if
40:	end function
41:	$P\left[6\right]$ = [7, 31, 127, 8191, 131071, 524287]

continued on following page

Table 3. Continued

Pseudo code	
42:	$S\left[6\right]$ = [0, 3, 8, 15, 28, 45]
43:	$A\left[6\right]$ = [random(|u|, P_0), random(|u|, P_1), …, random(|u|, P_5)]
44:	$L\left[P_5\right]$ = [0, 0, …, 0]
45:	CALL <<< 1 >>>Step1(P, S, A, L)
46:	$PHT\left[2^{64}, 2^{32}\right]$ ← concurrent hash table with default value of - 1
47:	CALL <<< k >>>Step2($P, S, A, PHT, Pattern$)
48:	$TH\left[n\right] = \left[0, 0, \ldots 0\right]$
49:	CALL <<< n >>>Step3($P, S, L, TH, Text$)
50:	CALL <<< 1 >>>Step4(TH) ← done using CUB library
51:	CALL <<< n >>>Step5($P, S, PHT, TH, Match$)

- In step2, we calculate each pattern's hash values in parallel and load their values into the hash table, leading to a total complexity of $O\left(\frac{k * m * h}{\tau}\right)$, where τ is the available number of threads and h is the number of hashes to be calculated.
- In step3, we calculate hash values for every character of the input text x in parallel, leading to a total complexity of $O\left(\frac{n * h}{\tau}\right)$.
- In step4, the prefix sum for every position of the hash values array is calculated using "*decoupled look-back*" algorithm if enough threads are available, it can have a complexity of $O\left(\log_2\left(n\right) * h\right)$, but since it's infeasible to have as much as $\frac{n}{2}$ threads, the total complexity will be calculated as $O\left(\frac{n * h}{\tau}\right)$.
- In step 5, for every position in the array we compute the six hash values, where we can compute each value in $O\left(1\right)$, and we check if the combination of the six hash values exists in the already built hash table or not. this can be done in $O\left(1\right)$ on average case, leading to a total complexity of $O\left(\frac{n * h}{\tau}\right)$.

Combining all the complexities together our total complexity will be $O\left(\frac{k * m * h}{\tau} + \frac{n * h}{\tau}\right)$ and since k, m and h are relatively small it's safe to say that total complexity is $O\left(\frac{n}{\tau}\right)$, which is near perfect.

RESULTS

In our experiments, we considered the number of patterns $k = 2^0, 2^2, 2^4, 2^6, 2^8$ patterns with $m = 10, 20, 30$ characters each. The input text x has $n = 2^{27}$ characters ($\approx$ 128Mbytes).

The input string x and the k patterns $Pattern_0, Pattern_1, \ldots, Pattern_{k-1}$ are randomly generated over the alphabet size $u = 2$.

Table 4 illustrates the execution time in milliseconds for our implementation for 4Parallel Rabin-Karp with String Validation Skipped execution times under normal conditions.

Table 5 shows speedups gained by comparing the execution times of the three different implementations Serial Rabin-Karp (SRK), Parallel Rabin-Karp (PRK), and finally Parallel Rabin-Karp with String Validation Skipped ($PRKSV$). The table shows speedup gains in each size of the input in normal conditions.

As shown, PRK is approximately 700 times faster than SRK except for the case when $k > 2^4$ and $m = 10$ speed up tends to vary a lot, and execution times of both implementations PRK and PRK increases significantly.

The reason for that is the significant increase of matches when k increases, because for alphabet size $u = 2$, and pattern length $m = 10$, we are left with 2^{10} possible unique patterns only and for instance when $k = 2^8$ it's expected there's 1 in 4 chances of a match.

Meanwhile, $PRKSVS$ is at least 150 times faster than SRK and its execution times are fairly stable with 0.01 variance with the disadvantage that it's 3-5 times slower than PRK because of the manipulation of the six hashing functions at once.

Table 4. Parallel Rabin-Karp with String Validation Skipped execution times under normal conditions

Number of patterns (k)	Length of each input pattern (m)		
	10	20	30
2^0	3.52	4.58	4.57
2^2	3.89	4.58	4.57
2^4	4.78	4.59	4.59
2^6	6.11	4.66	4.61
2^8	6.32	4.71	4.70

Table 5. Speedup in execution times under normal conditions

Comp.	$\frac{SRK}{PRK}$			$\frac{SRK}{PRKSVS}$			$\frac{PRKSVS}{PRK}$		
$k \setminus m$	10	20	30	10	20	30	10	20	30
2^0	764.2	700.8	691.4	150.4	180.5	177.1	5.08	3.88	3.90
2^2	706.9	700.8	700.2	152.7	179.5	177.9	4.63	3.90	3.93
2^4	617.1	703.7	701.5	165.0	178.2	179.2	3.74	3.95	3.91
2^6	597.3	688.8	704.9	202.4	180.2	182.6	2.95	3.82	3.86
2^8	993.6	690.0	693.6	347.7	181.6	181.9	2.86	3.80	3.81

The reason why someone would use the slower *PRKSVS* instead of the fast *PRK* is that an attacker can easily stall *PRK* by sending a packet full of patterns saved in the *PRK* system.

This DOS attack can be done easily because the attacker can install *PRK* into his/her system and gain access to the patterns database to construct the stalling packets.

DOS Attack

To test *PRKSVS* and *PRK* against DOS attack possible, we used the same input text x size which is 2^{27} characters ($\approx$ 128Mbytes) but changed the alphabet size to be $u = 1$ to assume worst case, meaning that there's only a single unique input pattern that will be matching in all packets.

Below are the execution times in milliseconds for the three different implementations.

We see that under a DOS attack, an attacker can stall both *SRK* and *PRK* and increase their execution time by up to 10,000% more than under normal conditions wherein *PRKSVS* , as we can see that its execution time is increased by 7% more than under normal conditions.

GPU 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 three implementations.

CONCLUSION

Our research has led to some significant advancements in the field of Network Intrusion Detection Systems (NIDS). Our innovative modification of the Rabin-Karp algorithm has demonstrated both reliable and stable performance when subject to load testing, effectively eliminating the threat of Denial of Service (DoS) attacks.

Through a combination of theoretical and practical testing, we've proven that our modified algorithm doesn't result in any misclassifications, thereby ensuring that false alarms triggered by potential attackers are effectively eradicated. This eliminates a major concern for many network security systems.

In terms of scalability, our modified Rabin-Karp algorithm excels, demonstrating excellent adaptability for parallel implementation. This is particularly crucial in our current digital landscape, where the need to process large amounts of data simultaneously is becoming increasingly essential.

Further reinforcing the strength of our modified Rabin-Karp algorithm, we have provided clear evidence that neither false positives nor DoS attacks can occur. This further highlights the robustness and reliability of our approach to enhancing network security.

A particularly noteworthy feature of our modification is the ability to handle an increased number of patterns or rules without significantly affecting the overall running time of the algorithm. This is possible because our approach involves converting all patterns into hashes, thereby eliminating the need for the original patterns. This stands in stark contrast to other approaches, where an increase in the number of patterns is a severe limitation, leading to considerable inefficiencies.

Overall, our work represents a significant stride forward in the improvement of Network Intrusion Detection Systems. We have modified the Rabin-Karp algorithm that not only provides a consistently high-performing and secure solution but also offers excellent scalability and the ability to efficiently

Table 6. Execution times for all implementations of Rabin-Karp under DoS attack

Impl.	*SRK*			*PRK*			*PRKSVS*		
$k \setminus m$	**5000**	**1000**	**1500**	**5000**	**1000**	**1500**	**5000**	**1000**	**1500**
2^0	212970	414719	615360	125.37	210.60	296.12	18.38	18.76	18.76

manage an increased number of patterns. These findings open new possibilities for further research and improvements in network security.

AUTHOR'S DECLARATIONS AND STATEMENTS

Compliance With Ethical Standards

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.

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.

Research Data Policy and Data Availability

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

Ahmad, Z., Shahid Khan, A., Wai Shiang, C., Abdullah, J., & Ahmad, F. (2021). Network intrusion detection system: A systematic study of machine learning and deep learning approaches. *Transactions on Emerging Telecommunications Technologies*, *32*(1), e4150. doi:10.1002/ett.4150

Aho, A. V, & Corasick, M. J. (1975). *Efficient String Matching: An Aid to Bibliographic Search*.

Aigbe, P., & Nwelih, E. (2021). Analysis and Performance Evaluation of Selected Pattern Matching Algorithms. *NIPES Journal of Science and Technology Research*, *3*(2). Advance online publication. doi:10.37933/nipes/3.2.2021.7

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

Baloi, A., Belean, B., Turcu, F., & Peptenatu, D. (2023). GPU-based similarity metrics computation and machine learning approaches for string similarity evaluation in large datasets. *Soft Computing*. Advance online publication. doi:10.1007/s00500-023-08687-8

Boukebous, A. A. E., Fettache, M. I., Bendiab, G., & Shiaeles, S. (2023). A Comparative Analysis of Snort 3 and Suricata. *2023 IEEE IAS Global Conference on Emerging Technologies (GlobConET)*, (pp. 1–6). IEEE. doi:10.1109/GlobConET56651.2023.10150141

Boyer, R. S., & Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, *20*(10), 762–772. doi:10.1145/359842.359859

Çelebi, M., & Yavanoğlu, U. (2023). Accelerating Pattern Matching Using a Novel Multi-Pattern-Matching Algorithm on GPU. *Applied Sciences (Basel, Switzerland)*, *13*(14), 8104. doi:10.3390/app13148104

Chen, C. L., & Lai, J. L. (2023). An Experimental Detection of Distributed Denial of Service Attack in CDX 3 Platform Based on Snort. *Sensors (Basel)*, *23*(13), 6139. doi:10.3390/s23136139 PMID:37447987

Fernandez De Arroyabe, I., Arranz, C. F. A., Arroyabe, M. F., & Fernandez de Arroyabe, J. C. (2023). Cybersecurity capabilities and cyber-attacks as drivers of investment in cybersecurity systems: A UK survey for 2018 and 2019. *Computers & Security*, *124*, 102954. doi:10.1016/j.cose.2022.102954

Groth, T., Groppe, S., Pionteck, T., Valdiek, F., & Koppehel, M. (2023). Hybrid CPU/GPU/APU accelerated query, insert, update and erase operations in hash tables with string keys. *Knowledge and Information Systems*, *65*(10), 4359–4377. doi:10.1007/s10115-023-01891-w

Gulyás, O., & Kiss, G. (2023). Impact of cyber-attacks on the financial institutions. *Procedia Computer Science*, *219*, 84–90. doi:10.1016/j.procs.2023.01.267

Hnaif, A., Jaber, K., Alia, M., & Daghbosheh, M. (2021). Parallel scalable approximate matching algorithm for network intrusion detection systems. *The International Arab Journal of Information Technology*, *18*(1), 77–84. doi:10.34028/iajit/18/1/9

Jakim, B., Shankar, D., Victo, G., George, S., Naidu, J., & Madhuri, S. (n.d.). Deep Analysis of Risks and Recent Trends Towards Network Intrusion Detection System. In *IJACSA) International Journal of Advanced Computer Science and Applications, 14*(1). www.ijacsa.thesai.org

Karcioglu, A. A., & Bulut, H. (2021). Improving hash-q exact string matching algorithm with perfect hashing for DNA sequences. *Computers in Biology and Medicine*, *131*, 104292. doi:10.1016/j.compbiomed.2021.104292 PMID:33662682

Karp, R. M., & Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, *31*(2), 249–260. doi:10.1147/rd.312.0249

Keserwani, P. K., Govil, M. C., & Pilli, E. S. (2023). An effective NIDS framework based on a comprehensive survey of feature optimization and classification techniques. *Neural Computing & Applications*, *35*(7), 4993–5013. doi:10.1007/s00521-021-06093-5

Knuth, D. E., & Morris James, H. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, *6*(2), 323–350. doi:10.1137/0206024

Li, Y., & Liu, Q. (2021). A comprehensive review study of cyber-attacks and cyber security; Emerging trends and recent developments. *Energy Reports*, *7*, 8176–8186. doi:10.1016/j.egyr.2021.08.126

Liao, H. J., Richard Lin, C. H., Lin, Y. C., & Tung, K. Y. (2013). Intrusion detection system: A comprehensive review. In Journal of Network and Computer Applications, 36(1), 16–24. doi:10.1016/j.jnca.2012.09.004

Merrill, D. (n.d.). *CUB: A pattern of "collective" software design, abstraction, and reuse for kernel-level programming*. Research Gate.

Merrill, D., & Garland, M. (n.d.). *Single-pass Parallel Prefix Scan with Decoupled Look-back*. Research Gate.

Mijwil, M., Unogwu, O. J., Filali, Y., Bala, I., & Al-Shahwani, H. (2023). Exploring the Top Five Evolving Threats in Cybersecurity: An In-Depth Overview. *Mesopotamian Journal of Cyber Security*, 57–63. 10.58496/MJCS/2023/010

Najam-ul-Islam, M., Zahra, F. T., Jafri, A. R., Shah, R., Hassan, M., & Rashid, M. (2022). Auto implementation of parallel hardware architecture for Aho-Corasick algorithm. *Design Automation for Embedded Systems*, *26*(1), 29–53. doi:10.1007/s10617-021-09257-7

Nunes, L. S. N., Bordim, J. L., Ito, Y., & Nakano, K. (2020). A rabin-karp implementation for handling multiple pattern-matching on the gpu. *IEICE Transactions on Information and Systems, E103D*(12), 2412–2420. 10.1587/transinf.2020PAP0002

Papadogiannaki, E., Ioannidis, S., & Tsirantonakis, G. (n.d.). *Network Intrusion Detection in Encrypted Traffic Article Social media analysis during political turbulence View project Network Intrusion Detection in Encrypted Traffic*. Research Gate. https://www.researchgate.net/publication/362263511

Sardar, A., Issa, A., & Albayrak, Z. (n.d.). DDoS Attack Intrusion Detection System Based on Hybridization of CNN and LSTM. In Acta Polytechnica Hungarica, 20(2).

Siva Kumar, C., Sandeep, V., & Reddy, K. (n.d.). Design of Acceptance Sampling based Network Intrusion Detection system using Deep Learning Techniques Mohd khalid 2. In Journal of Survey in Fisheries Sciences, 10(1).

Sommestad, T., Holm, H., & Steinvall, D. (2022). Variables influencing the effectiveness of signature-based network intrusion detection systems. *Information Security Journal*, *31*(6), 711–728. doi:10.1080/19393555.2021.1975853

Anas Annas, A seasoned Software Engineer specialising in distributed systems, currently making significant contributions at Workday in the distribute compute frameworks department. With a background that includes impactful roles at AWS, Noon, HackerRank, and Dell Boomi, Also was a Lecturer at Ain Shams University.

Mahmoud Fayez, Received the BSc. degree in Computer Systems from the University of Ain Shams, Egypt, in 2007, and the MSc in Computer Science from Ain Shams University, Egypt, in 2014. In 2008, Joined the Department of Computer Systems at Ain Shams university as a Teaching Assistant. He became associate lecturer in 2014. His current research interests include CUDA, MPI, OpenMP, Machine Learning, Computer Visison and Elastic Optical Network.

Heba Khaled, is an esteemed computer scientist and Associate Professor at Ain Shams University. With a triumphant academic journey including a Bachelor's, Master's, and Ph.D. in Computer Science, specializing in computer systems, she has become a respected authority in the field.