A Study on Signature-Based Web API Intrusion Detection Algorithm Using Regular Expressions*

Youngjae Kim[1], I Wayan Adi Juliawan Pawana[1], Junyong Lee[2], Myeongcheol Kim[2], Bonam Kim[1], and Ilsun You[1]

Kookmin University, Seoul, Republic of Korea[1]
Penta Security, Seoul, Republic of Korea[2]
{zeroash1225, adijuliawan, kimbona, isyou}@kookmin.ac.kr[1]
{junyong, mckim}@pentasecurity.com[2]

Abstract

With the rapid development of the Internet, the use of APIs(Application Programming Interface) has become increasingly prevalent, highlighting the growing importance of API security. According to the OWASP(Open Worldwide Application Security Project) API Security Top 10 and Traceable's API security risk reports, the current state of API security threats is evident. In response to these challenges, this paper proposes a signature-based web API intrusion detection algorithm utilizing Regular Expressions(Regex). Intrusion detection can be categorized into two approaches: signature-based and specification-based. The signature-based approach defines attack patterns in advance and identifies attacks by matching incoming API calls with these predefined patterns. To effectively capture these patterns, this study employs Regular Expressions to establish a set of rules for string matching. The proposed algorithm converts API calls into structured JSON(JavaScript Object Notation) format and uses Regular Expression rules to identify potential malicious patterns within the data. Finally, this paper concludes by introducing the F1 score as one of the methods to evaluate the performance of the proposed algorithm and by suggesting future research directions focused on a hybrid approach that combines signature-based detection with behavior-based or machine learning methods.

Keywords: Web API Security, Anomaly Detection, Signature Based Detection, Regular Expression

1 Introduction

An API (Application Programming Interface) serves as a boundary between different parts of a software system, enabling one part to use functionalities from another system or component through a series of operations [1]. With the advancement of the internet and the acceleration of digital transformation following the Fourth Industrial Revolution, APIs have become a key element in modern digital ecosystems, essential for both software development and business operations. Enterprises and developers leverage APIs to facilitate complex system interactions and rapidly introduce new functionalities. APIs have evolved beyond simple data transmission tools, enabling seamless collaboration and integration among various software systems, contributing to continuous innovation and maintaining competitive advantage for businesses. APIs have gained even more prominence with the development of cloud computing, mobile applications, and the IoT(Internet of Things). Additionally, they have emerged as crucial tools in realizing corporate automation. Many companies utilize APIs to automate internal processes and various tasks such as customer service, data processing, and supply chain management. This not

*Proceedings of the 8th International Conference on Mobile Internet Security (MobiSec'24), Article No. 16, December 17-19, 2024, Sapporo, Japan.

only improves operational efficiency and reduces costs but also enables the swift processing of large-scale tasks that are difficult for humans to manage manually. This automation plays a critical role in helping companies adapt to fast-changing market environments and maintain their competitiveness. APIs also act as catalysts for collaboration between companies, beyond being mere software development tools. By exposing their APIs, companies enable external developers and partners to use them to build new applications, thus opening up opportunities for new business models and revenue generation. This API ecosystem has become a driving force behind the growth of many digital platforms today. A prime example is the Google Maps API[2], which has been widely adopted across various industries, fueling the advancement of location-based services. Another notable example is the Twitter API[3], which has facilitated the development of social media analytics tools and marketing platforms.

However, as the significance of APIs grows, security concerns have become a critical issue. Since APIs serve as gateways for external connections, any vulnerabilities can make them prime targets for attackers. In fact, many cyberattacks exploit API vulnerabilities, leading to severe consequences such as data breaches, service disruptions, and reputational damage. A research report by Traceable on the current state of API security risks[4] highlights the increasing frequency of API-related data breaches. According to the report, 60% of organizations experienced an API-related breach in the past two years, with 74% of them encountering at least three instances of API-related data leakage. This suggests that attackers are repeatedly exploiting known vulnerabilities. To address these challenges, the Open Worldwide Application Security Project (OWASP)[5], a community dedicated to supporting the design, development, and operation of reliable applications, has released the API Security Top 10-2023[6]. This updated version, the second since its initial release in 2019, outlines common risk factors in API security and provides guidelines to raise awareness of API vulnerabilities.

Designing and operating secure and reliable APIs is a significant challenge for all organizations. This issue transcends technical security concerns, encompassing legal compliance and maintaining customer trust. Given the growing importance of APIs and their associated security issues, a detailed discussion on this topic is timely. In this paper, we propose a signature-based API intrusion detection algorithm that leverages Regular Expressions(Regex) to address these concerns. By predefining attack signatures and using Regular Expressions to detect or validate these patterns within API call strings, we aim to detect API attacks.

2 Related Work

APIs facilitate interactions between software systems, and their importance continues to grow. Given that APIs can expose sensitive internal information to external entities, their vulnerabilities have become prime targets for attackers.

One of the most prominent methods for enhancing API security is the use of Intrusion Detection Systems (IDS), which can be categorized into signature-based detection and behavior-based detection. Signature-based detection techniques define patterns of previously known attacks and detect real-time matches in API calls. A well-known example of such a system is Snort, which detects known attack patterns in network traffic. While signature-based systems demonstrate high detection accuracy for recognized threats, they may show vulnerabilities when dealing with novel or obfuscated attacks. In contrast, behavior-based detection methods analyze deviations from learned patterns of normal traffic. Recent research has utilized machine learning techniques to analyze anomalous API call behavior and detect potential threats[7]. Although this approach can detect new attack types, it suffers from a high rate of false positives, presenting the challenge of improving accuracy.

Among these intrusion detection methods, this paper focuses on signature-based detection, which predefines attack patterns and matches them with API calls for verification. Regular Expressions can be employed in this context. Regular Expressions are powerful tools for identifying specific patterns within strings and are widely used in text analysis and pattern matching. Recent studies have attempted to apply Regular Expressions to detect security threats in APIs. For instance, research utilizing Regular Expressions for SQL injection detection has demonstrated its effectiveness in blocking attacks with distinct patterns[8]. Regular Expressions are well-suited for detecting predefined attack patterns within the text of API calls. By leveraging this technique, the efficiency of pattern matching in signature-based detection systems can be enhanced, enabling the near-instantaneous detection and prevention of potential attacks through API calls, ensuring that threats are addressed within milliseconds of the API request being made.

A systematic literature review on scalable hardware implementations of pattern matching algorithms[9] highlights the significance of regular expression matching, particularly in improving the performance of IDS through approaches like DFA and NFA. FPGA-based solutions have been developed to enhance the performance of regular expression matching in deep packet inspection, such as the Reinhardt architecture, which supports real-time updates of regex patterns with high throughput[10]. The SAID method integrates signature-based detection with AI-powered deep analysis to address the limitations of detecting novel attacks and handling large-scale network traffic[11]. Improvements to Snort, a widely used intrusion detection system, focus on enhancing packet capture capabilities and optimizing the detection engine using the high-performance Hyperscan regex engine for better handling of high-speed network traffic[12]. Regular Expressions have proven effective in handling undecided string formats, such as in advertisements filtration using various Regex libraries[13]. The Enhanced HES method improves pattern matching for regular expressions by supporting complex SP-Free patterns while optimizing the regex handler check procedure[14]. Our research focuses on a software-based approach using Regular Expressions (Regex) for efficient signature-based intrusion detection in dynamic API environments, aiming for flexible and lightweight detection of malicious patterns directly within API traffic. Unlike existing studies that emphasize hardware acceleration or deep learning models for general network traffic analysis, this paper specifically addresses the gap in providing an effective and adaptable solution for Web API security, which has been less explored in prior research. Despite these advancements, there remains a gap in effectively addressing Web API security using a lightweight IDS specifically designed to detect malicious API calls. This paper aims to address this gap by focusing on the unique challenges of Web API intrusion detection, providing a tailored solution for enhanced API security.

Consequently, in this paper, we propose a signature-based API intrusion detection algorithm that utilizes Regular Expressions to address API security vulnerabilities and enhance overall protection.

3 Method

The Figure 1 presents an algorithm designed for the real-time security analysis of API traffic. It begins by mirroring traffic from the API Gateway, capturing incoming requests for further scrutiny. These captured API calls are then parsed into a structured format, extracting details such as the host and URI information, which are essential for subsequent analysis. The core of the system lies in its detection engine, which uses a series of predefined rules to identify potential security threats. These rules are stored as signatures, which are processed by a regex rule parser, translating them into regular expression patterns. The parsed rules are then applied

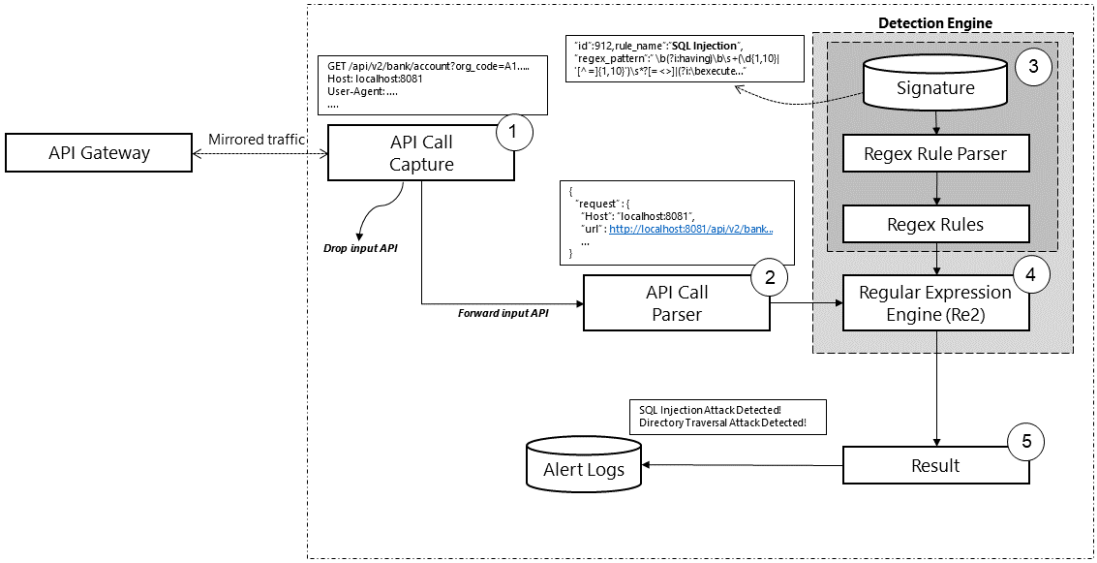

Figure 1: Signature-based web API detection algorithm using regular expressions

by a Regular Expression Engine (Re2), which matches the incoming API requests against these patterns to detect possible attacks, such as SQL injection or directory traversal attempts. When a suspicious pattern is detected, the system generates an alert log, documenting the nature of the detected threat. This process allows security administrators to stay informed about potential attacks and take necessary actions. Overall, this algorithm aims to provide a robust mechanism for monitoring and safeguarding API interactions against various types of cyber threats.

Algorithm 1 Capture API Call and Preprocessing

Input: API Call from API Gateway
Output: Processed API Call

```
DataAPI ← List of Data API
input_API ← API Call from API Gateway
input_API ← Preprocess API Call and extract URI in API Call
if input_API is in DataAPI then
    forward input_API
else
    drop input_API
end if
```

Algorithm 1 filters incoming API calls and delivers them. The algorithm begins by initializing a list called *'DataAPI'*. This list contains a collection of pre-approved or whitelisted APIs that the system is permitted to process. Next, the algorithm receives an incoming API call from the API gateway. This call is stored in a variable named *'input_API'*. The algorithm then preprocesses the received *'input_API.'* The URI (Uniform Resource Identifier) is extracted from the API call during preprocessing. This URI is critical because it identifies the specific resource or service the API call intends to access or interact with. The algorithm checks if

the preprocessed *'input_API'* (particularly its URI) matches any entries in the *'DataAPI'* list. This step is a filtering mechanism to ensure that only recognized and authorized API calls are processed further. It is considered valid if the *'input_API'* is found in the *'DataAPI'* list. The algorithm then forwards this API call for further processing or execution. It is deemed unrecognized if the *'input_API'* does not match any entries in the *'DataAPI'* list. Consequently, the algorithm drops this API call, meaning it is not processed further and is effectively discarded. This mechanism effectively prevents potentially harmful or irrelevant requests from reaching the detection engine. Importantly, when an API call is dropped, no alert log is generated. This is a deliberate design choice aimed at reducing noise in the logging system, ensuring that only significant events are recorded. The decision-making process involved in this step is crucial, as it enhances the overall efficiency and effectiveness of the API intrusion detection system by focusing on relevant data.

Algorithm 2 Parse API Call

Input: API Call (Raw)
Output: API Call (JSON Format)

1: input_API ← API Call from API Gateway in Raw Format
2: input_API_JSON_Format ← Parse Raw API Call to JSON Format
3: **return** input_API_JSON_Format

Algorithm 2 describes a straightforward algorithm that converts an incoming raw API call into a structured JSON[15] format. This conversion is essential for processing the API call consistently and standardized. The algorithm begins by receiving an API call in its raw format from the API Gateway. This raw format typically includes unstructured or semi-structured data, which may not be immediately suitable for processing. The received raw API call is stored in a variable named *'input_API'*. The algorithm's core function is to parse the raw API call into a structured JSON format. This process involves analyzing the raw data, extracting relevant information, and organizing it into a JSON object. The result of this parsing operation is stored in a variable named *'input_API_JSON_Format'*. After successfully parsing the raw API call, the algorithm outputs the structured data by returning the *'input_API_JSON_Format'*. This JSON format is now ready for further processing, analysis, or forwarding within the system.

Algorithm 3 describes an algorithm designed to initialize and generate a set of Regular Expression rules from a database of known attack signatures. These regex rules identify potentially malicious patterns in data, such as in incoming API calls. The algorithm starts by initializing an empty list or collection called *'regex_rules'*. This collection will eventually hold the regex patterns derived from the database of known attack signatures. The algorithm then loads a database containing signatures of known attacks. These signatures represent patterns associated with malicious activities and are stored in a variable named *'database_signature'*. This database is the source from which the regex rules will be created. The algorithm enters a loop until all entries in the *'database_signature'* have been processed. During each iteration, the algorithm reads one signature rule pattern from the database, which is stored in a variable called *'signature_rule_pattern'*. The algorithm attempts to parse each *'signature_rule_pattern'* read from the database. Parsing involves interpreting the pattern and converting it into a regex format that can be used for pattern matching. If the parsing is successful, the algorithm converts the signature rule pattern into a regex rule, storing it in a variable named *'regex_signature_rule'*. If the parsing is not successful, the algorithm output the error If the parsing is successful, the

Algorithm 3 Initialization of Regex Rules

Input: Database of Known Attack Signatures
Output: Regex Rules

```
initialize empty regex rules
database_signature ← Load Database of Known Attack Signatures
while not end of database_signature do
    signature_rule_pattern ← Read the signature rule from the database
    if parse(signature_rule_pattern) is success then
        regex_signature_rule ← parse(signature_rule_pattern)
        regex_rules ← add regex_signature_rule
    else
        alert parse error for signature_rule_pattern
    end if
end while
save regex rules in memory
return regex rules
```

resulting *'regex_signature_rule'* is added to the *'regex_rules'* collection. This step ensures that each valid signature pattern is stored as a regex rule for future use. The loop continues until all signature patterns in the *'database_signature'* have been processed and either added to the *'regex_rules'* collection or skipped due to parsing errors. After all the signature rules have been processed, the *'regex_rules'* is saved in memory. This allows the system to access and utilize these rules for real-time pattern matching quickly. Finally, the algorithm returns the *'regex_rules'* collection. This collection is now ready to detect known attacks based on the regex patterns.

Algorithm 4 Match API Call with Attack Signature in Regex

Input: API Call, Regex Rules
Output: Matching Result

```
flag ← false
matching_result ← false
API Call ← load API Call
regex_rules ← load Regex Rules from memory
for each regex signature rule in regex rules do
    if match(regex signature rule, API Call) == true then
        flag ← true
        matching_result ← regex signature rule
        break
    end if
end for
return matching_result
```

Algorithm 4 outlines an algorithm designed to match an incoming API call against a set of predefined regex rules representing known attack signatures. The goal is to determine whether the API call contains patterns indicative of malicious activity. The algorithm starts by initializing two variables:

- *'flag'* is set to 'false'. This variable acts as a signal to indicate whether a match is found.

- *'matching_result'* is also set to 'false'. This variable will hold the result of the matching process, specifically the regex rule that matches the API call if a match is found.

The algorithm loads the incoming API call, which needs to be evaluated against the regex rules. This API call is stored in the variable *'API_Call'*. Next, the algorithm loads the set of regex rules from memory. These rules represent patterns associated with known attack signatures and are stored in the variable *'regex_rules'*. The algorithm initiates a loop over each *'regex_signature_rule'* in the *'regex_rules'* collection. The purpose of this loop is to check whether any of these regex rules match the content of the *'API_Call'*. For each *'regex_signature_rule'*, the algorithm attempts to match it against the *'API_Call'*. The 'match' function returns 'true' if the *'regex_signature_rule'* matches a pattern in the *'API_Call'*. If a match is found, then:

- The *'flag'* variable is set to 'true', indicating that a match has been found.

- The *'matching_result'* is updated to hold the *'regex_signature_rule'* that matched the *'API_Call'*.

- The algorithm then breaks out of the loop, as it has found a match and does not need to continue checking other rules.

If no match is found after checking all the regex rules, the loop ends naturally, and the *'matching_result'* remains 'false'. Finally, the algorithm returns the *'matching_result'*. If a match was found, *'matching_result'* will contain the specific *'regex_signature_rule'* that matched the API call; otherwise, it will return 'false', indicating no match was detected.

Algorithm 5 Save Alert Log to Database

Input: Matching Result
Output:

```
1: if matching result is not false then
2:     LoadedAlertLogs ← load database alert logs
3:     UpdatedAlertLogs ← save matching result to database
4: end if
```

Algorithm 5 outlines a simple algorithm designed to save an alert log to a database whenever a matching result, indicating a potential security threat, is detected. This ensures that any identified suspicious activity is recorded for future reference or analysis. The algorithm begins by receiving an input called *'matching_result'*. This input represents the outcome of a previous process (such as an API call matching against attack signatures) that determines whether a suspicious pattern has been detected. The algorithm first checks whether the *'matching_result'* is not 'false'. This condition means a match was found, and the result contains relevant information about the suspicious activity. If the *'matching_result'* is valid (i.e., not 'false'), the algorithm loads the existing *'LoadedAlertLogs'* from the database. This data contains all previously recorded alerts, which the algorithm will update with the new matching result. The algorithm then updates the *'UpdatedAlertLogs'* by adding the new *'matching_result'* to the database. This step ensures that the identified security threat is logged and stored for future analysis or action.

4 Evaluation

In this study, we employ precision, recall, and the F1 score as the primary metrics to evaluate the performance of our API intrusion detection algorithm. These metrics are particularly useful for assessing the effectiveness of detection algorithms in environments with imbalanced data, such as API traffic, which may exhibit diverse characteristics. This section explains the definitions and calculations of these metrics and presents the procedure for evaluating the algorithm's performance based on them. To evaluate the performance of the API intrusion detection system, it is necessary to use a dataset containing various API requests, including both legitimate and malicious ones. The dataset should also reflect a wide range of attack types, such as SQL injection, cross-site scripting, authentication bypass, privilege escalation, and more. Publicly available datasets that can be leveraged for this purpose include CICIDS2017[16] or NSL-KDD[17] which include network-based attack scenarios. These datasets offer a rich set of API traffic scenarios with various attack types, enabling comprehensive testing of the proposed detection algorithm's performance under diverse conditions. For the purposes of this study, we plan to employ such datasets to experimentally validate the algorithm's effectiveness. The results of the classification made by the algorithm will be categorized into four possible outcomes by Table 1

Classification	**Description**
True Positive (TP)	Malicious requests that are correctly identified as intrusions.
False Positive (FP)	Legitimate requests that are incorrectly flagged as intrusions.
True Negative (TN)	Legitimate requests that are correctly identified as non-intrusions.
False Negative (FN)	Malicious requests that are not detected and are incorrectly classified as legitimate.

Table 1: Classification outcomes for the API intrusion detection algorithm

These classification outcomes provide the foundation for calculating precision, recall, and the F1 score, which are defined as follows:

- Precision: Precision represents the proportion of correctly identified intrusions among all requests flagged as intrusions by the algorithm. It is defined as:

$$\text{Precision} = \frac{TP}{TP + FP}$$

  Precision focuses on minimizing false positives by measuring how many of the detected intrusions are truly malicious.

- Recall: Recall, also known as sensitivity or the true positive rate, measures the proportion of actual intrusions that were correctly identified. It is defined as:

$$\text{Recall} = \frac{TP}{TP + FN}$$

  Recall emphasizes minimizing false negatives, as it indicates how many actual intrusions were detected by the algorithm.

- F1 Score: The F1 score is the harmonic mean of precision and recall, providing a balanced measure that accounts for both metrics. It is defined as:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1 score is particularly useful when the data is imbalanced, as it balances the trade-off between precision and recall, ensuring that neither metric is disproportionately favored.

In this study, we will use the aforementioned metrics, with a particular emphasis on the F1 score, to comprehensively evaluate the algorithm's performance. The evaluation procedure is outlined as follows:

- **Dataset Preparation:** A dataset comprising both legitimate and malicious API requests will be applied to the algorithm.
- **Classification Output:** For each API request, the algorithm will classify it as either an intrusion (positive) or a non-intrusion (negative). The classification results will be compared against the true labels to calculate the numbers of TP, FP, TN, and FN.
- **Precision and Recall Calculation:** Using the formulas defined above, the precision and recall will be calculated based on the classification results.
- **F1 Score Calculation:** The F1 score will be computed using the calculated precision and recall, providing a single metric that reflects both aspects of the algorithm's performance.

The F1 score is a critical metric for evaluating the performance of intrusion detection systems, particularly in imbalanced data environments. It is especially relevant in the context of API intrusion detection, where the number of malicious requests may be relatively small compared to legitimate requests. This study proposes a method for balanced evaluation of the performance of an API intrusion detection algorithm that detects API attacks, using precision, recall, and the F1 score.

5 Conclusion

In this paper, we proposed a signature-based web API intrusion detection algorithm utilizing Regular Expressions to address the growing security risks posed by API vulnerabilities. Our approach leverages Regular Expressions to match predefined attack signatures with incoming API calls, allowing for precise detection of known threats such as SQL injection, Cross-Site Scripting, and Directory Traversal. The key advantage of this method lies in its efficiency and adaptability. Regular Expressions provide a flexible framework for pattern matching, allowing the system to be updated with new attack signatures as they emerge. This makes it particularly useful for real-time environments where rapid detection of known threats is critical. However, the system's reliance on predefined signatures limits its ability to detect novel or obfuscated attacks, highlighting a key challenge for signature-based detection systems. To overcome this limitation, future work should explore hybrid approaches that combine signature-based detection with behavior-based or machine learning methods. These approaches could enhance the system's ability to detect previously unknown threats while maintaining the strengths of signature-based detection. Furthermore, expanding the testing environment to include more diverse datasets and utilizing the F1 score proposed in this paper would enable a clearer evaluation of the algorithm's performance in real-world API traffic scenarios. In conclusion, while the proposed algorithm offers a robust solution for detecting known API threats, further development is needed to address the evolving nature of API security challenges. By integrating this signature-based approach with other detection methods, we can build a more comprehensive and adaptive defense against the wide range of attacks targeting APIs today.

Acknowledgement: This work was supported by Penta Security under the 2024 Kookmin University-Penta Security Information Security Industry-Academic Cooperation Program(Project Number: S2024-0081).

References

[1] Neil Madden. *API Security IN ACTION*. Manning, 2020.

[2] Google. Google maps platform. https://developers.google.com/maps.

[3] X. X developer platform. https://developer.x.com/en/docs/x-api.

[4] Traceable. 2023 state of api security: A global study on the reality of api risk. Technical report, Ponemon, 2023.

[5] OWASP. About the OWASP. https://owasp.org/about/.

[6] Erez Yalon, Inon Shkedy, Paulo Silva. OWASP API Security Top 10. Technical report, OWASP, 2023.

[7] Ayesha S. Dina and D. Manivannan. Intrusion detection based on machine learning techniques in computer networks. *Internet of Things*, 16:100462, 2021.

[8] R Senthan, EYA Charles, and SR Kodituwakku. Regular expressions based sql injection detection. 2021.

[9] Malik Imran, Faisal Bashir, Atif Raza Jafri, Muhammad Rashid, and Muhammad Najam ul Islam. A systematic review of scalable hardware architectures for pattern matching in network security. *Computers & Electrical Engineering*, 92:107169, 2021.

[10] Jaehyun Nam, Seung Ho Na, Seungwon Shin, and Taejune Park. Reconfigurable regular expression matching architecture for real-time pattern update and payload inspection. *Journal of Network and Computer Applications*, 208:103507, 2022.

[11] Hoang V. Vo, Hanh P. Du, and Hoa N. Nguyen. Ai-powered intrusion detection in large-scale traffic networks based on flow sensing strategy and parallel deep analysis. *Journal of Network and Computer Applications*, 220:103735, 2023.

[12] Longwen Shuai and Suo Li. Performance optimization of snort based on dpdk and hyperscan. *Procedia Computer Science*, 183:837–843, 2021. Proceedings of the 10th International Conference of Information and Communication Technology.

[13] Jianshan Li, Dazhong He, Fang Liu, and Huan Wang. The application of regex in advertisements filtration and performance analysis. In *2016 8th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, volume 01, pages 28–32, 2016.

[14] Mohammad Hashem Haghighat and Jun Li. Toward fast regex pattern matching using simple patterns. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 662–670, 2018.

[15] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017.

[16] Canadian Institute for Cybersecurity. CICIDS 2017 Dataset. https://www.unb.ca/cic/datasets/ids-2017.html, 2017. Accessed: 2024-10-01.

[17] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A. Ghorbani. A detailed analysis of the KDD CUP 99 data set. In *Proceedings of the Second IEEE Symposium on Computational Intelligence for Security and Defense Applications*. IEEE, 2009.