Design and Development of A Cloud-Based IDS using Apache Kafka and Spark Streaming

Leon Wirz, Rinrada Tanthanathewin, Asipan Ketphet, Somchart Fugkeaw
Sirindhorn International Institute of Technology, Thammasat Univerity, Pathum Thani, Thailand
leon.wir@dome.tu.ac.th, rinrada.tanth.gal@gmail.com, asipanketphet@gmail.com, somchart@siit.tu.ac.th

Abstract—**Owing to the efficient resource management, accessibility, and high service availability, cloud computing has been leveraged by several intensive-data processing applications such as big data analytics, social media applications. These applications are typically based on the development of web service and web application. Even though web-based technology offers effective communication and implementation, it has been susceptible to various kinds of attack. In this paper, we investigate possible attacks on REST which is a commonly used protocol for the web service implementation. In REST, HTTP requests are mapped to GET, POST, PUT, and DELETE that have been proven to be prone to common attacks including Automated Brute Forcing on web-based login, HTTP flood attacks, SQL injections (SQLi), and Cross-Site Scripting (XSS). To this end, we propose a design and implementation of the cloud-based IDS to detect such attacks by employing Apache Kafka and Spark streaming to classify and process the high volume of user inputs in REST HTTP communication. To detect the anomalous inputs, we apply the signature-based approach to construct an IDS engine based on a set of known attack patterns that will be leveraged by the Spark Streaming. Specifically, we introduce a new string comparison collection that improves the False Positive (FP) rate in SQL injection detection, which has been a major issue in most proposed IDS currently available. In our experiment, the system is able to determine malicious patterns with high performance as well as to generate SMS alerts and log the event in a Google Cloud Storage Bucket in an efficient manner.**

Keywords—Intrusion Detection, Apache Kafka, Spark Streaming, SQL injection, Cross-site scripting, Brute Force Login, HTTP flooding

I. INTRODUCTION

An intrusion detection system (IDS) is a security system which main functionality is the monitoring and analysis of system events for the purpose of finding and providing real-time or near real-time alerts and warnings of attempts of unauthorized access to system resources. Protocol-based intrusion detection systems are specialized IDSs that are commonly installed on web servers and used to exclusively monitor and analyze HTTP or HTTPS requests in a stream-like format. Typically, there are three types of intrusion detections methods, which are as follows:

i) Signature/Heuristic-based detection, which is a simple and efficient method of detection that involves the matching against large collections of rules or known patterns of malicious data. Nevertheless, the ability of an IDS that uses this method to correctly detect an intrusion solely depends on the available collection of signatures. Hence, the system cannot detect unknown intrusions in which patterns are not contained in the collection of signatures.

ii) Anomaly-based detection, which involves actual observations is more dynamic and more intelligent compared to the former method. Using either statistical methods or machine-learning are used to analyze events and categorize them into either being a legitimate request or an intrusion attempt. Statistical methodologies are applied to observe the behavior of events using univariate, multivariate, or time-series models to observe measurable metrics. The Machine-learning approach uses Classification ML models, which work by intaking events and activities and classifying them into being either intrusion attempts or not, as has been done in [1]. Additionally, more advanced ML models can classify events into more than two classes as in [2], where Siva Reddy and Saravanan have developed multiple classification models with more than 2 variations of outputs using one-hot encoding. Regardless of the mentioned advantages, compared to signature-based approach, the anomaly-based detection's performance is not as favorable and has some negative effects on less powerful machines, running real-time IDS.

iii) Distributed/Hybrid-based detection, which is performed using a combination of Signature/Heuristic-based detection and Anomaly-based detection. This detection method has a more complex structure, but with it comes a better identification rate and response time.

Essentially, implementing IDS on cloud is non-trivial as the system needs to deal with various formats and massive volumes of data or traffics. In addition to the reliability of its detection mechanism, the system performance and scalability issue are also the crucial factors for the design of a cloud-based IDS. In addition, the system systems should give results in a real-time manner, must not take up a lot of resources, have a low false positive and false negative rate, have high to permanent uptime, and cover a wide area of intrusion methods that it can detect. Generally, existing cloud-based IDS systems have capability to handle high volume of data by blending the system on its network with mirrored VMs. In this paper, we propose an alternative solution in getting closer to the raw data based on the system features of Kafka and Spark Streaming. Our IDS system is designed to seamlessly work with these data processing tools to deliver higher capability in processing and detecting common attacks as well as offer faster detection. In our proposed system, Kafka [3] acts as the receiving end of a bridge that allows all the streaming data to pass through via four separate topics in a broker, being GET, POST, PUT, and DELETE, which correspond to their actual HTTP method name. These four topics, which are streaming data are then sent to separate Spark Streaming jobs that each detect their own Intrusions namely *Automated Brute Forcing on web-based login, HTTP flood attacks, SQL Injections (SQLi), and Cross-Site Scripting (XSS).*

Automated Brute Forcing [4] on web-based login is the act of using a program or script to forcefully send login requests, typically GET requests in HTTP to a web server. The frequency of requests sent varies from program to program and machine sent from. The Automated Brute Force Login detection algorithm alerts and logs requests if a certain request

2022 19th International Joint Conference on Computer Science and Software Engineering (JCSSE) | 978-1-6654-8510-4/22/$31.00 ©2022 IEEE | DOI: 10.1109/JCSSE54890.2022.9836264

limit threshold is breached from a single IP address attempting to send requests at a high frequency. HTTP flood attacks are similarly detected using a different algorithm. Since HTTP flood attacks originate from all four HTTP request methods, being GET, POST, PUT, and DELETE, all requests are being processed in the HTTP flood attack detection algorithm. The frequency counting part also remains the same with the exception of the request frequency limit which is dependent on the implementation of IDS itself. As for SQL injections and Cross-Site Scripting detection, both have algorithms that function closely. A list of sub-strings for each attack is constructed which are extracted from datasets from [5] and [6], which are most likely to occur in an attack attempt.

Spark Streaming [7] is a framework that works on Spark; therefore, it utilizes parallel computation and synchronization of multiple machines on one or more clusters. Dividing streaming data into batches by time frames allows the usage of batch processing, furthermore the creation of a specialized Data structure called Resilient Distributed Dataset (RDD). Algorithms created in Spark typically run MapReduce Jobs which have been proven to give tremendous amounts of resource utilization and efficiency [8].

Based on the capabilities of Apache Kafka and Spark streaming, we calibrate them to construct as an assisted platform for our cloud-based IDS. This paper's proposed system implements an SMS API, being Vonage's SMS API [9], embedded into every Spark Streaming Job, with the aim of sending an SMS alert to each phone number passed as arguments during the initialization of each Spark Job. This will make sure that after each batch's completion, if one or many attack attempts have been detected, SMSs will be sent out to each phone number containing the Time Stamp, IP address, and what attack attempt has been detected. This information is used for querying more details from the Cloud Storage Bucket. Hence our system implementation involves the adoption of partial attributes of each related work in order to achieve an optimal IDS with the addition of real-time alerts. Moreover, a new technique for creating strings that are part of the collections which are used to be compared in the SQL injection detection algorithm is being used to reduce the FP rate. The said collection consists of improved strings that have been adjusted to take advantage of SQL's syntax that needs spaces to be considered valid for certain language variations. Moreover, our proposed system can be used as a foundation for more specialized implementation in the future.

II. Related Work

In 2020 Hai and Khiem [10] proposed architectures for processing IDS logs using Spark Streaming. The focus is to improve the performance of Network Intrusion Detection Systems (N-IDS) by using distributed processing and parallel computing with Apache Spark. Three distributed computing models are compared with each other, being: Distributed Snort, HBase & Impala, and HBase & Impala with Additional VMs. The computational efficiency is best when HBase and Impala are implemented with additional VMs. This inspired the usage of using a distributed system to efficiently process the great number of requests that are poured into the system.

This concept is then further built on by the experimentation in [8], which is used to evaluate the performance and has presented promising results when it comes to huge amounts of records. In this system, 100 000, 300 000, and 500 000 records have been used as a testing load for a 1-interval time frame. These records are sent to Spark streaming via Apache Kafka. The results of the experimentations indicate that performance issues do not occur even if the number of records fed into the system is immense.

Recently, Elmasry, Akbulut, and Zaim [11] proposed a design of integrated cloud-based intrusion detection systems (CIDS) using third-party cloud service. This design integrates all the modules that are used in an IDS to be migrated to a third-party cloud environment, where the operations of monitoring, processing, analysis, prediction, and response are accomplished on the cloud. Elmasry's, Akbulut's, and Zaim's modules have been partially implemented and mapped into this paper's proposed system. The Monitoring Module is depicted as the Kafka broker, which receives data streams from the four producers depending on the HTTP request method. The Processing and Analysis Modules are mapped to the four Spark Jobs which contain algorithms to detect intrusions. The final module this paper implement is the Alert module which consists of an embedded Vonage API call for each Spark Job accompanied by the logs that are saved on a dedicated Google Cloud Storage bucket.

Our implementation is heavily influenced by an Open-Source GitHub project made by Xu [12]. Visor which is a real-time log monitoring program using Kafka is used to log HTTP requests in a special format. Apache Kafka takes in or digests these requests from producers to topics according to HTTP request methods and then relays them to consumers that specifically request these topics.

Debnath et al [13] have conceived a model for Real-time Log Analysis System called LogLens. The Dynamic Programming Algorithm is used to detect abnormal log sequences of an event or transaction. This algorithm goes through different parts of a log's entry and checks important parameters against a deterministic value, string, pattern, or Wildcard. This idea is applied in this paper's algorithm for detecting SQL injections and cross-site scripting, where collections of Wildcards are used to compare patterns or signatures that fall under each category.

The detection mechanism is placed in a manner where the actual request is not forwarded to the system except for Cross-Site Scripting which is affecting the front-end part of the architecture of an application. This design choice has been proposed by Joshi, Ravishankar, Raju, and Rave [14] in 2017. In [14], it is explained that rather than having requests that may contain SQL injections being sent directly to a database, it will give a higher probability of preventing or taking action short after an attack if the system can give an output fast enough. The second part of the proposed system also includes the insertion of cross-site scripting detection before loading into the requested web page. Comparatively to the SQL injection detection's module position, Cross-site Scripting detection can be accomplished before the new page has loaded in. The reason remains the same for the position of the SQL injection detection module.

Nevertheless, all the aforementioned approaches have not addressed the ability to alert personnel who takes care after the web application of occurring intrusion attempts, while having a low False Positive rate in detecting SQL injections and Cross-site Scripting. These problems are tackled by this paper's proposed architecture. We also conducted the experiments with varied request load sizes.

III. Technological Background

Kafka is a distributed streaming platform that is highly scalable, fault-tolerant, and allows a high level of parallelism and decoupling between data producers and data consumers. Nowadays it is considered an industry standard when it comes to near real-time to real-time processing of streaming data. Meaning that Kafka is a critical component of most Big Data Platforms and hance the Hadoop ecosystem. Kafka consists of brokers which are Kafka nodes on a cluster and topics which are categories of streams and streaming records that can be partitioned and replicated and also support multiple writers (producers) and readers (consumers). Each broker has its own topic(s) and partition. The producer can push updated data into Kafka corresponding to its topic. In this case, the producer that produces messages and sends them to Kafka is the webserver and topics are GET, POST, PUT, and DELETE. The consumer can pull the updated data that the producer has pushed earlier from Kafka, which in this paper are the four Spark streaming jobs that detect intrusions.

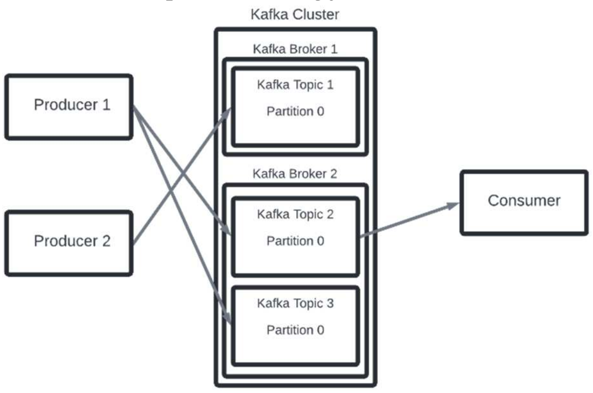


Figure 1 : Kafka Structure

Spark Streaming [7] is a framework used for large-scale stream processing. It achieves second scale latencies, that is the delay that should be minimized in seconds/milliseconds creating a near real-time-to-real-time processing time. Spark streaming requires a cluster to run spark jobs the same way Spark is running jobs. The major difference to Spark is that Spark streaming receives data in a stream formula, which in this paper's case is from a Kafka Topic. The received streaming data is split by a duration. The streaming data in each duration is then considered as a Batch, which afterwards can be used in Spark as a normal Batch. Batch operations of spark, such as Transformations that create new RDDs and Actions which return a result to the driver program are applied to RDDs in order to create the desired output.

Resilient distributed datasets (RDDs) are immutable and partitioned collections of objects or data. The immutability of RDDs makes these datasets unchangeable and hence does not create any synchronization problems from updating and overwriting. Instead, Spark creates a new RDD when a Transformation operation is used on an RDD. Additionally, the partitioning of RDDs allows the distributed computation when operations are called and guarantees no conflicts due to RDDs being immutable.

In our proposed system, four separate Spark streaming jobs are used of which all run on a Google Dataproc cluster [15]. The previously mentioned Jobs are Brute Force Job, HTTP Flood Job, SQLi Job, and XSS Job. Each of these jobs receives its streaming data from Kafka, which sends data to jobs according to their subscribed topic. This means that each Spark Streaming Job can connect to a Kafka broker and then choose to subscribe to one or multiple Kafka topics in that broker.

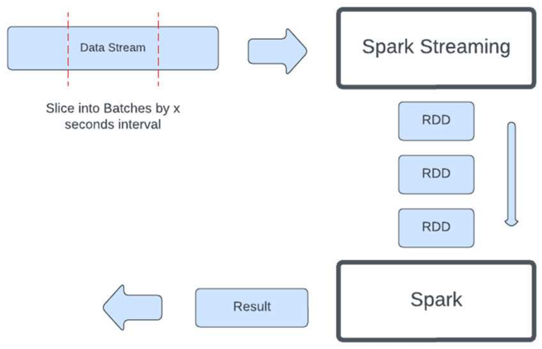
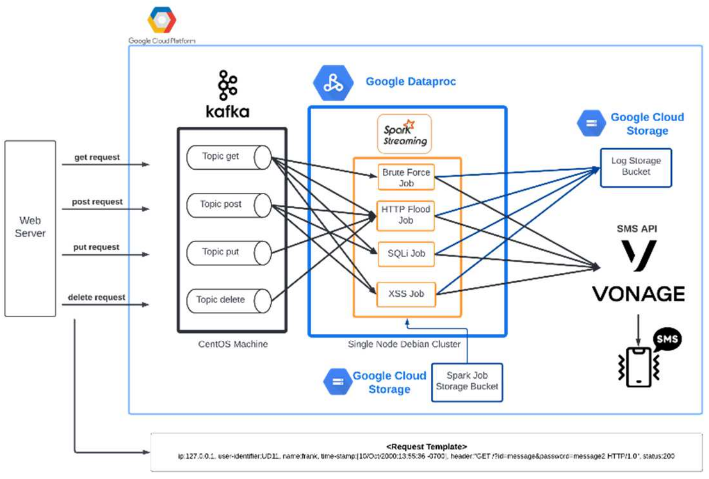


Figure 2 : Spark Streaming Flow

IV. Our Proposed System

Figure 3 illustrates the overall system architecture and shows how our proposed system works. The system itself is entirely assembled on Google Cloud Platform [16], starting from HTTP requests which are directly received from the web server by Apache Kafka. The code below shows a request template received by Apache Kafka. This request pattern has been modified from Xu's Visor system [12] log format since it contains all the important information needed in identifying intrusions. In addition, for future modifications, more information can be added to the HTTP request pattern depending on the web application's vulnerabilities that depend on that system itself.



<Request Template>

```
ip:127.0.0.1, user-identifier:UD11, name:frank, time-stamp:[10/Oct/2000:13:55:36 -0700], header:"GET /?id=message&password=message2 HTTP/1.0", status:200
```

Figure 3 : Our Proposed System Model

Apache Kafka then digests HTTP requests in the form of four topics: GET, POST, PUT, and DELETE, depending on the HTTP method type of each request. The HTTP request as a string is then sent to Spark Streaming, being the consumer of the topics, running on a Google Dataproc cluster [15]. Google Dataproc allows users to create and run jobs on different cluster types and versions. A single node cluster in this case provides a single node that acts as both the master and a worker. For the used request format sent to Kafka, the following template is required:

ip:127.0.0.1, user-identifier:UD11, name:frank, time-stamp:[10/Oct/2000:13:55:36 -0700], header:"GET /?id=message&password=message2 HTTP/1.0", status:200

Spark Streaming then runs four isolated jobs: Automated Brute Forcing on web-based login job, HTTP flood attacks job, SQL Injections (SQLi) job, and Cross-Site Scripting (XSS) job. For the first few steps, all algorithms have the same pattern. Assuming that in the sent format from the Web application, each field of each request is separated by a comma and each request is separated by a new line. We split the request stream by the new line delimiter "\n" first, which will separate the requests. The individual requests will then be split up by using a comma as the delimiter, while the request's 0th index, being the IP, will be mapped to the 4th index, which is derived from the user input.

1) Automated Brute Forcing on web-based login detection Algorithm

With an Automated Brute Forcing on web-based login attempt, the system will detect exploits with the GET HTTP method, meaning that this Spark streaming job subscribes to Kafka's GET topic. Using the flatmap method the request stream is split into different parts, where the argument being *n* is used to split each line as its own data.

```
flatmap(_.value().split("\n"))
```

```
(ip:127.0.0.1, user-identifier:UD11,
name:frank, time-stamp:[10/Oct/2000:13:55:36 -0700],
header:"GET /?id=message&password=message2
HTTP/1.0", status:200)

(ip:127.0.0.2, user-identifier:UD12,
name:john, time-stamp:[10/Oct/2000:13:56:01 -0700],
header:"GET /?id=message&password=message2
HTTP/1.0", status:200)
```

The algorithm counts the number of times an IP address calls the HTTP GET method in a particular time period. If the specified threshold is exceeded, it is considered a brute force attack and the algorithm saves the IP address and the number of attempts in a log file according to [17] [18]. Due to brute force login attempts varying in frequency and velocity as expressed in [19], where an attacker can use different hardware modifications to achieve an extreme number of attempts per second quota. But an amateur attacker might use more unsophisticated tools where the frequency of attempts may not be as high. Hence the threshold for determining what a brute force login attempt is, may not be the focus of this paper.

```
map(rdd => (rdd.split(", ")(0), 1)).reduceByKey((x, y)
=> x+y).filter(x => x._2 > requestThreshold)
```

```
(ip:127.0.0.1, 15)
(ip:199.12.26.22, 16)
(ip:149.244.168.229, 22)
(ip:42.40.79.231, 65)
(ip:219.57.228.72, 101)
(ip:3.111.203.205, 11)
(ip:29.13.130.4, 69)
(ip:39.119.21.208, 42)
```

2) HTTP flood detection Algorithm

For the HTTP flood attack detection algorithm, the system detects attempts of attacks through the GET, POST, PUT, and DELETE HTTP methods, meaning this algorithm is subscribed to every Kafka topic. Similar to the Automated Brute Forcing detection algorithm, first the requests streams are split into different parts. Secondly, the algorithm filters malicious IP addresses that have a number of attempts that exceed the limited threshold. Finally, the malicious attempts are saved into the log file saving location which is a Google Cloud Storage bucket.

3) SQL Injection (SQLi) detection Algorithm

With the SQL Injection (SQLi) detection algorithm, the system is attacked based on the GET and POST HTTP methods, so the algorithm subscribes and filters only those two Kafka topics. First, the algorithm must rearrange the request stream into a form of request strings that consists of the IP address as a Key and header part as a Value in the MapReduce architecture by using both the flatmap and map method.

```
flatmap(_.value().split("\n")).map(x => (x.split(", ")(0),
x.split(", ")(4)))
```

```
(ip:127.0.0.1, header:"GET /?id=admin" or
"1"="1&password=message2 HTTP/1.0")
(ip:127.0.0.1, header:"GET
/?id=message1&password=message2 HTTP/1.0")
(ip:29.13.130.4, header:"POST /?id=" or 1=1 –
&password=message2 HTTP/1.0")
(ip:149.244.168.229, header:"POST /?id= or 0=0 --
&password=message2 HTTP/1.0")
(ip:127.0.0.1, header:"GET
/?id=message5&password=message2 HTTP/1.0")
(ip:127.0.0.1, header:"GET
/?id=message6&password=message2 HTTP/1.0")
```

The algorithm filters the request strings that match SQLi request patterns which are found in the self-made list of hand-picked strings that are contained in most of the SQLi payloads from the dataset in [16] and save them into a log file.

```
reduceByKey((x, y) => x + " , " + y).filter(x =>
sqli_payload_list.exists(y => x._2.contains(y)))
```

```
(ip:127.0.0.1, header:"GET /?id=admin" or
"1"="1&password=message2 HTTP/1.0", .... )
(ip:3.111.203.205, header:"GET /?id=' and
substring(password/text(),1,1)='7&password=message2
HTTP/1.0")
(ip:29.13.130.4, header:"POST /?id=" or 1=1 –
&password=message2 HTTP/1.0")
```

The list or collection that is used consists of used keywords that are part of commonly used SQL injection payloads [5]. This system however has adjusted the way that these keywords are stored in the collection. Many SQL frameworks have a limitation of needing whitespace between keywords, and many of the SQL injection payloads do have whitespace between each word used. Therefore, this collection consists of strings that contain spaces encapsulating the keywords such as OR, AND, UNION, etc. This greatly reduces the FP rate which is a rather common problem in most IDSs.

Collection sql_payload_list = {" or ", ")or(", " and ", ")OR(", " OR ", " AND ", " UNION ", " mid(", " int(", " elt(", " cast(", "table_name", " null "...}

4) Cross-Site Scripting (XSS) detection Algorithm

For the Cross-Site Scripting (XSS) detection algorithm, the system will detect attempts of attacks via the GET and POST HTTP methods, hence this algorithm subscribes to and filters only those two Kafka topics. Similar to the SQL Injection detection algorithm, first, the algorithm rearranges the request streams into the specific. After that, the algorithm filters malicious IP addresses that have a request pattern match with an XSS payload pattern in the self-made collection of commonly used strings that are part of XSS payloads in [6]. Finally, the algorithm saves all malicious attempts into a log file on the Google Cloud Storage bucket.

5) SMS Alerting

Each of the four algorithms will have a filter method that is called at the end. The algorithm will first check if the final RDD that is created from the filter transformation method is not empty. After doing so a new SMS message will be created and sent to all phone numbers that were passed during the Spark Job's initialization. Therefore, if no attempts have been detected, SMSs messages will not be created and sent.

V. EXPERIMENT

This paper's experimentation to acquire results includes the usage of a Request Generation script which is used for generating an immense number of requests to simulate real-world applications. Each Spark Streaming job is performed using four amounts of requests, being 10 000, 25 000, 50 000, and 100 000 requests respectively. The environment this system is tested on includes a single node cluster for Apache Kafka, where the node consists of a single core CPU and 3.75GB of memory. The Dataproc cluster for running Spark jobs is also a single node cluster consisting of a quad core CPU and 15GB of memory, while the used Spark version is 3.1 and the Kafka version is 3.1. The total processing time for each job can be seen in Figure 4, while it should be noted that efficiency is not the main essence of this paper.

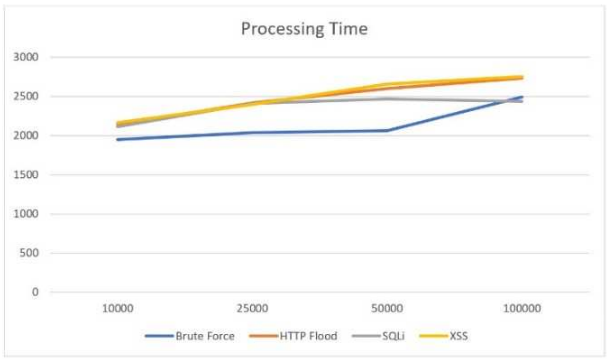


Figure 4: Spark Jobs Processing Time

The pattern of the plotted graph in Figure 4.1 goes hand-in-hand with Spark's advantage of being able to cache RDDs for efficient reuse and MapReduce's benefit of parallelization. The resulting graph shows that the processing time is between 2 and 2.8 seconds, where the number of requests is between 10 000 and 100 000. The time difference compared to the difference of the requests is minuscule to the point that it can be presumed that the rate of the processing time will increase in the same manner until the number of requests reached the limit of the computational power of the cluster.

Furthermore, we can determine the throughput of each Spark Streaming Job which we calculate as the value of Requests per Milliseconds. Once again, the results follow the theory of Spark which can be seen in Figure 5. As the number of Requests increases but the processing time remains close to the processing time of fewer requests, the throughput skyrockets. The graph shows that for all Spark Streaming Jobs the throughput builds up with the number of requests, hence proving that resource utilization is strikingly high when the amount of data that is processed is high, which correlates to real-world Web Applications.

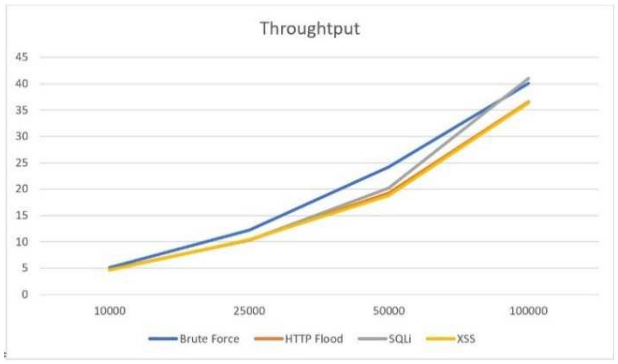


Figure 5 : Spark Jobs Throughput.

Table 1 shows the detection confusion matrix for both the SQL injection detection algorithm and the Cross-Site scripting detection algorithm. The results convey the quality of the signature lists for both algorithms as the TP numbers for all cases are prominently high. For all the request numbers, the percentage of the actual detected attacks compared to the total number of attacks averages at 89.38% for SQL injection and 97.5% for Cross-Site Scripting.

Table 1 : SQLi and XSS detection confusion matrix.

	TP	TN	FP	FN
10 000				
SQLi	86	9 888	12	14
XSS	96	9 900	0	4
Total	100 Attacks			
25 000				
SQLi	169	24 781	19	31
XSS	200	24 800	0	0
Total	200 Attacks			
50 000				
SQLi	260	49 694	6	40
XSS	299	49 699	1	1
Total	300 Attacks			
100 000				
SQLi	250	99 745	5	0
XSS	237	99 750	0	13
Total	250 Attacks			

Moreover, the number of false alarms is relatively low, where there are on average 5 false alarms per 10 000 requests for SQL injection detection and 1 false alarm per
40 000 requests for Cross-Site Scripting. The reason for the big difference in false detection rate is the way SQL, which is used in SQL injections, is closer to Human Language than JavaScript, HTML, and CSS, which are used in Cross-Site Scripting. Additionally, the sample spaces that are used are the SQL-injection-payload-list and XSS-payload-list constructed by I. Tasdelen in [5], which contains the most used SQLi and XSS payloads. As for the SQLi signature collection, most if not all signatures used for checking, take advantage of whitespaces and brackets that are required in SQL syntax. Hence giving the low false positive rate, due to the need of Spaces or Brackets, which will not be present in Legitimate user input as often. Finally, the last number that is important being the FN rate, meaning attacks that are not being noticed or detected by the system, which is relatively low for all cases. On average SQL injections that are not being detected relate to 8 attacks per 10,000 requests and for Cross-Site Scripting the FN rate amounts to 5 attacks per 10 000 requests. This correlates with this Protocol IDS being a Signature-based IDS, where one big disadvantage is that the system cannot detect intrusions that are not described in the signature collection.

Finally, the SMS alert system has also been tested by using four different phone numbers based in Thailand. All four phone numbers were able to receive all SMSs that were sent, where the message bodies contained the sufficient information that is needed to query the logs from the Cloud Storage bucket.

VI. Final remarks

In this paper, a protocol IDS hosted on a cloud environment has been designed and developed. The background information for the architecture has been explored in-depth, which included Apache Kafka and Spark Streaming, how they work, and are implemented in this paper's proposed system design. The main intrusions the proposed system detects include Brute-force login attacks, HTTP flooding attacks, SQL injections, and Cross-site scripting. Each of the attacks' patterns have been explored and algorithms to detect them in Spark Streaming have been developed in order to match each attack's patterns, thus making this IDS a Signature-based IDS. To test and evaluate the system, multiple and varying sizes of requests have been created using a python script that matches the system's request pattern. Each Spark streaming job has been tested using with various requests per batch. The SMS alerting system works as intended where SMS messages are correctly formatted and sent. Additionally, a new method of filtering SQL injection payloads has been deduced, where spaces are taken into consideration when wildcards are constructed. This approach can be applied to future implementations, where such patterns can be identified in other intrusion methods such as more advanced Cross-Site scripting attacks, XML external entity (XXE) injections, and OS command injections.

References

[1] J. K. R, S. Balaji B, N. Pandey, P. Beriwal and A. Amarajan, "An Efficient SQL Injection Detection System Using Deep Learning," 2021 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE), 2021, pp. 442-445, doi: 10.1109/ICCIKE51210.2021.9410674.

[2] S. V. Siva reddy and S. Saravanan, "Performance Evaluation of Classification Algorithms in the Design of Apache Spark based Intrusion Detection System," 2020 5th International Conference on Communication and Electronics Systems (ICCES), 2020, pp. 443-447, doi: 10.1109/ICCES48766.2020.9138066.

[3] "Documentation," *Apache Kafka*. [Online]. Available: https://kafka.apache.org/documentation/. [Accessed: 22-May-2022].

[4] T. Kakarla, A. Mairaj and A. Y. Javaid, "A Real-World Password Cracking Demonstration Using Open Source Tools for Instructional Use," 2018 IEEE International Conference on Electro/Information Technology (EIT), 2018, pp. 0387-0391, doi: 10.1109/EIT.2018.8500257.

[5] I. Tasdelen, "PAYLOADBOX/SQL-injection-payload-list: SQL Injection Payload List," *GitHub*, 2021. [Online]. Available: https://github.com/payloadbox/sql-injection-payload-list. [Accessed: 22-May-2022].

[6] I. Tasdelen, "Payloadbox/XSS-payload-list: Cross site scripting (XSS) Vulnerability payload list," *GitHub*, 2021. [Online]. Available: https://github.com/payloadbox/xss-payload-list. [Accessed: 22-May-2022].

[7] "Spark overview," *Overview - Spark 3.2.1 Documentation*. [Online]. Available: https://spark.apache.org/docs/latest/. [Accessed: 22-May-2022].

[8] M. T. Tun, D. E. Nyaung and M. P. Phyu, "Performance Evaluation of Intrusion Detection Streaming Transactions Using Apache Kafka and Spark Streaming," 2019 International Conference on Advanced Information Technologies (ICAIT), 2019, pp. 25-30, doi: 10.1109/AITC.2019.8920960.

[9] "Communications APIs," *Vonage.* [Online]. Available: https://www.vonage.com/communications-apis/. [Accessed: 22-May-2022].

[10] T. H. Hai and N. T. Khiem, "Architecture for IDS Log Processing using Spark Streaming," 2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE), 2020, pp. 1-5, doi: 10.1109/ICECCE49384.2020.9179188.

[11] Elmasry, Wisam, Akbulut, Akhan and Zaim, Abdul Halim. "A Design of an Integrated Cloud-based Intrusion Detection System with Third Party Cloud Service" Open Computer Science, vol. 11, no. 1, 2021, pp. 365-379. https://doi.org/10.1515/comp-2020-0214

[12] W. Xu, "Xuwenyihust/visor: A real-time Apache log monitor using Kafka & Spark Streaming, with fake log generator.," *GitHub*, 2017. [Online]. Available: https://github.com/xuwenyihust/Visor. [Accessed: 22-May-2022].

[13] B. Debnath et al., "LogLens: A Real-Time Log Analysis System," 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018, pp. 1052-1062, doi: 10.1109/ICDCS.2018.00105.

[14] P. N. Joshi, N. Ravishankar, M. B. Raju and N. C. Ravi, "Contemplating Security of Http From SQL Injection and Cross Script," 2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), 2017, pp. 1-5, doi: 10.1109/ICCIC.2017.8524376.

[15] "Dataproc | Google Cloud," *Google*. [Online]. Available: https://cloud.google.com/dataproc/. [Accessed: 22-May-2022].

[16] *Google*. [Online]. Available: https://cloud.google.com/. [Accessed: 22-May-2022].

[17] L. Bošnjak, J. Sreš and B. Brumen, "Brute-force and dictionary attack on hashed real-world passwords," 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2018, pp. 1161-1166, doi: 10.23919/MIPRO.2018.8400211.

[18] M. M. Najafabadi, T. M. Khoshgoftaar, C. Calvert and C. Kemp, "Detection of SSH Brute Force Attacks Using Aggregated Netflow Data," 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), 2015, pp. 283-288, doi: 10.1109/ICMLA.2015.20.

[19] Keonwoo Kim, "Distributed password cracking on GPU nodes," 2012 7th International Conference on Computing and Convergence Technology (ICCCT), 2012, pp. 647-650.