TÉCNICO LISBOA

CorrForce: A Scalable Live Traffic Correlation System

João Pedro Antunes Aragonez

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Nuno Miguel Carvalho dos Santos

Examination Committee

Chairperson: Prof.ª Valentina Nisi
Supervisor: Prof. Nuno Miguel Carvalho dos Santos
Member of the Committee: Prof. Diogo Miguel Barrinha Barradas

October 2024

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

Firstly, I express my deepest gratitude to my advisor, Professor Nuno Santos, for welcoming me as his student, believing in my potential, and for his enthusiastic guidance throughout this work. His invaluable insights and continuous support have been instrumental in shaping this thesis. His mentorship has inspired me to pursue excellence in my research endeavors. I extend my thanks to Daniel Castro, who was always promptly available for guidance and discussion. His expertise and willingness to assist at any moment provided clarity and direction during critical phases of this project, having significantly contributed to the completion of this thesis. Special thanks to Daniela Lopes for her support in the early stages of this work, when exploring previous research. I am also grateful to INESC-ID for the financial support through this last few months. Their funding and resources provided the necessary infrastructure and tools to conduct this study effectively. I owe my deepest gratitude to my family, particularly my parents, for their endless support, encouragement, and care over all these years. Their belief in my potential and their continuous motivation have been a driving force behind my academic and personal growth. Lastly, I would like to thank all my friends and colleagues for their support and camaraderie, as their encouragement and friendship have made this journey very enjoyable and memorable.

To all, my most sincere thank you.

This work was supported by national funds through IAPMEI via
the SmartRetail project (ref. C6632206063-00466847).

Abstract

The increasing use of anonymity and privacy-preserving network systems (e.g. Tor network, VPNs) presents significant challenges for authorities investigating unlawful communications at scale. One approach is to make use of correlation techniques to match client traffic against the traffic observed in malicious servers. However, there are some inherent challenges. These networks tend to be complex and usually lead to large volumes of exchanged data. Existing correlation models, while promising, are often limited by their focus on single use cases(i.e. Tor, VPNs) and reliance on static datasets, which restricts their real-world effectiveness. On top of this, such models are also tuned for specific datasets and do not capture the effects of changes in the Tor network. We propose CorrForce, a system that enhances live traffic correlation by enabling Internet Service Provider (ISP)s to actively capture and categorize traffic traces, enhancing the efficiency of correlation efforts such as Tor de-anonymization and VPN detection. This thesis demonstrates the effectiveness of CorrForce through a use case focused on Tor traffic correlation for de-anonymizing sessions between Tor clients and Onion Services, utilizing the SUMo correlation model. CorrForce also facilitates collaboration among ISPs by offering a novel, distributed, live, and query-based framework that can integrate different correlation models and uses industry-ready frameworks such as Kafka and Spark Structured Streaming. Key contributions of this thesis include the development of a comprehensive system architecture for CorrForce, the introduction of a new query language to specify flow correlation requests, and a prototype capable of executing such queries using SUMo over simulated Tor traffic. Preliminary results indicate a throughput of 35,000 pairs per second and close to linear scaling, showcasing CorrForce's potential and paving the way for real-time applications in Tor traffic correlation.
This work was supported by national funds through IAPMEI via the SmartRetail project (ref. C6632206063-00466847).

Keywords

Tor, Onion Services, Traffic correlation, Flow Correlation, Real-time Systems, Distributed Systems, De-anonymization, Anonymity Network

Resumo

A crescente complexidade dos sistemas de redes anónimas e privadas, juntamente com os vastos volumes de tráfego, apresenta desafios significativos para as autoridades que venham a procurar investigar comunicações em larga escala. Esta tese aborda tais desafios ao focar-se no tema da correlação de fluxo e tráfego, que aproveita diversas fontes de dados (e.g. pacotes de rede) para extrair informação relevante. Embora mecanismos de correlação existentes mostrem resultados promissores, estes muitas vezes são limitados por focarem-se em casos de uso únicos e por basearem os resultados em conjuntos de dados estáticos, o que restringe a sua eficácia no mundo real. O sistema proposto, CorrForce, promove a correlação de tráfego ao vivo, permitindo que os Provedores de Serviços de Internet (ISPs) capturem e classifiquem ativamente os dados de tráfego, aumentando a eficiência de modelos correlação, como a deanonimização de tráfego Tor e a detecção de VPN. Esta tese demonstra a eficácia do CorrForce por meio de um caso de uso focado na correlação de tráfego do Tor para deanonimizar sessões de cliente a Serviços Onion, utilizando a metodologia de correlação do *SUMo*. O CorrForce oferece uma nova estrutura de multi-correlação distribuída, sobre tráfego ao vivo e baseada em queries de consulta, que integra vários mecanismos como o Kafka e o Spark Structured Streaming, e facilita a colaboração entre ISPs. Algumas das principais contribuições desta tese incluem o desenvolvimento de uma arquitetura de sistema abrangente, a introdução de uma nova linguagem de consulta para manipulação eficaz de dados, e um protótipo capaz de executar pedidos de correlação utilizando o modelo *SUMo* em dados simulados do Tor. Resultados preliminares indicam um throughput de 35 000 pares/s e escalabilidade linear, demonstrando o potencial do sistema e abrindo portas para a sua aplicação em tempo real tanto no domínio atual como em futuros.

Palavras Chave

Tor, Serviços Onion, Correlação de tráfego, Correlação de fluxos, Sistemas em tempo real, Sistemas distribuídos, Deanonimização, Rede de Anonimato Tor

Contents

1 Introduction **1**

1.1 Motivation . . . 2

1.2 Contributions . . . 3

1.3 Organization of the Document . . . 4

2 Background **5**

2.1 Traffic Correlation Attacks . . . 5

2.2 Other Correlation Use Cases . . . 9

2.3 Desired Properties for Distributed Systems . . . 11

3 Related Work **13**

3.1 Big Data Processing Technologies and Techniques . . . 14

3.2 Distributed Query Processing . . . 16

3.3 Distributed Machine Learning . . . 18

3.4 Traffic Analysis and Networking . . . 19

3.5 Real-Time System Metrics . . . 20

4 Design **21**

4.1 CorrForce Architecture Overview . . . 22

4.2 Internet Service Provider (ISP) Probe . . . 23

4.3 Correlator . . . 28

4.4 User API . . . 31

4.5 CorrForce Limitations . . . 35

5 Implementation **39**

5.1 Implementation Overview . . . 40

5.2 CorrForce's Query Mechanism . . . 42

5.3 Live Tor Flow Generation and Capture over a Simulated Network . . . 45

5.4 Queuing Correlation Data through Apache Kafka . . . 47

5.5 Distributed Correlation Over Spark Structured Streaming . . . 49

6 Evaluation **59**

6.1 Evaluation Environment 60

6.2 Performance and Scalability 60

6.3 *SUMo* Reviewed Correlation Performance 67

6.4 Live Correlation 68

7 Conclusion **75**

7.1 Achievements 76

7.2 Future Work 76

Bibliography **77**

A CorrForce configurations **89**

A.1 Deployment over Docker 89

A.2 Distributed Queuing System (DQS) Kafka Cluster Configuration 90

List of Figures

2.1 The SUMo pipeline comprising two online phases – filtering and matching phase – and one offline training phase. . . . 8

4.1 Architecture overview. . . . 22
4.2 ISP probe flow capture pipeline. . . . 24
4.3 Correlator system architecture. . . . 28
4.4 Time diagram of query execution. . . . 29
4.5 Example Flow Request (FR) derived from the query use case from Figure 4.8. . . . 30
4.6 Example of pair descriptor. . . . 30
4.7 Query attributes. . . . 32
4.8 Query usage case 1. . . . 33
4.9 Query usage case 2. . . . 34
4.10 Query usage case 3. . . . 36

5.1 Implementation overview. . . . 40
5.2 CorrForce *Web UI*: New query page. . . . 42
5.3 CorrForce *Web UI*: Query table page. . . . 43
5.4 CorrForce *Web UI*: Query resulting pairs page. . . . 44
5.5 Simulating Client accessing Onion Service (OS) with ISP flow capture enabled. . . . 46
5.6 Spark cluster architecture. . . . 50
5.7 Complete CorrForce implementation. . . . 53

6.1 Flow join performance under $M \times M$ all-matching flows per stream. . . . 61
6.2 Flow join performance under $M \times M$ never-matching flows per stream. . . . 62
6.3 Full correlation pipeline performance comparison for 400 all matching flows of size 2000 packets between Sub-Set-Sum base and optimized version. . . . 63
6.4 Throughput while varying the number of packets per flow under the 400 all-matching flows input. 63
6.5 Throughput while varying the number of all-matching flows to correlate each with 2000 packets. 63

6.6 Throughput variation for different number of workers. 64
6.7 Ideal vs Practical Scale-Up factors from the baseline cluster of three worker node. 64
6.8 Response Time (RT) for a synthetic query over the OS Test dataset. 65
6.9 *SUMo*'s results with released code implementation. 66
6.10 *SUMo*'s results with paper algorithm implementation for the paper version. 67
6.11 Precision-Recall curve for Source and Target separation models. 68
6.12 Receiver operating characteristic (ROC) curve for Source and Target separation models. 68
6.13 Live correlation experiment query. 70
6.14 Precision-Recall curve for the correlation results with Ground Truth(GT) and Model Classified(MC) flows. 71
6.15 ROC curve for the correlation results with Ground Truth(GT) and Model Classified(MC) flows. . . 71

List of Tables

4.1 Probing Policy parameters. 25
4.2 *SUMo*'s flow descriptor features. 26

5.1 Spark application configuration. 51

6.1 Google Cloud resources per machine for static cluster evaluation stage. 60
6.2 OS Test query statistics. 65
6.3 Source separation statistics over realistic flows. 70
6.4 Target separation statistics over realistic flows. 70
6.5 Average response time for the live query. 72

A.1 Broker and Controller configuration. 90

Listings

5.1 “CorrForce Spark application main function.” 52
5.2 “SUMo correlation algorithm revised.” 54
5.3 “Sub-Set-Sum optimization.” 56

Acronyms

CNN	Convolutional neural network
DFS	Distributed File System
HDFS	Hadoop Distributed File System
ISP	Internet Service Provider
OS	Onion Service
AUC	Area Under Curve
LEA	Law Enforcement Agencies
FR	Flow Request
QH	Query Handler
CM	Cluster Manager
DQS	Distributed Queuing System
OSF	Onion Service Flow
CF	Client Flow
KRaft	Apache Kafka Raft
ISR	"In-Sync" Replica
RDD	Resilient Distributed Dataset
ROC	Receiver operating characteristic
TPR	True Positive Rate
FPR	False Positive Rate
RT	Response Time
PT	Pluggable Transport

1

Introduction

Contents

1.1 Motivation 2

1.2 Contributions 3

1.3 Organization of the Document 4

In the present days, the complexity of anonymity and privacy-preserving network systems, concerning the sophisticated architectures and large volumes of exchanged data [1], makes it difficult for authorities to investigate communications at scale. This highlights the essential role of flow/traffic correlation, where diverse data sources such as network packets can incorporate valuable information. To tackle the challenge of *correlating substantial volumes of live traffic data from diverse sources*, there is a necessity to design and implement a system capable of not only efficiently and effectively processing extensive amounts of online data, but also guaranteeing that multiple sources of traffic data, i.e. Internet Service Provider (ISP)s, actively participate in the process. With live correlation, it is intended to have ISPs actively capturing and categorizing groups of traffic data (i.e. a flow) to ensure consistent availability and enable on-demand correlation.

Collaborating with ISPs opens the door to applying traffic correlation techniques to a variety of widely used scenarios. Among these scenarios are the Tor de-anonymization [2], VPN traffic detection and classification [3,4], media piracy detection [5, 6], and anomaly detection [7, 8]. In this thesis, the focus relies uniquely on the Tor

de-anonymization case presented by Lopes et al. [2]. However, the broad range of available correlation models brings up the importance to create a versatile system, capable of supporting different correlation domains.

1.1 Motivation

Previous research regarding state-of-the-art correlation models [2, 3, 9] demonstrated relatively good results (i.e. achieving precision as good as 99%). However, these studies typically focus on correlating data from a single use case. We have not found any tool capable of integrating the efforts of multiple correlation models into a single system. Combining the functionality of multiple models into a single tool would simplify their use, reduce the need for multiple tools, and make it easier to compare the results of different models.

Furthermore, most current solutions follow a centralized approach and face limitations in processing live data. They require the collection of large data traces beforehand, which constrains their ability to handle and process information promptly. Additionally, these models were often tested on static, specific, and sometimes small datasets, neglecting aspects of traffic simulation and capture (e.g. traffic density) that could affect the final results. Therefore, testing in realistic and diverse scenarios is crucial for a comprehensive evaluation.

With this aspects in mind, developing a system capable of live traffic correlation has the potential to significantly enhance the efficiency and effectiveness of correlation models (e.g. Tor de-anonymization, VPN traffic detection and classification, etc.). The live aspect aims to speed up interactions with the system and achieve faster correlation goals by pre-establishing mutual agreements with ISPs. This approach would avoid bureaucratic delays and facilitate on-demand correlation through active traffic analysis. Fast results are crucial for identifying recent or ongoing events (e.g. DDoS attacks early detection, when performing anomaly detection correlation). Rapid identification of such threats allows for prompt action, reducing potential damage and maintaining network integrity.

To generalize the solution for accommodating multiple traffic correlation techniques, we must demonstrate the applicability and effectiveness of our approach. This is illustrated through a use case involving Tor traffic correlation for de-anonymizing client-Onion Service (OS) sessions. Specifically, we employ the method proposed by Lopes et al. [2], known as the *SUMo* Correlator. This use case will showcase the system's capability to handle complex traffic correlation scenarios.

Tor is a service capable of offering anonymity to both senders and receivers in network communication. Its utility concerns scenarios where communication privacy is essential [10], specifically in personal, medical, legal, governmental, or financial applications, all having the possibility of requiring a certain degree of privacy. However, the Tor anonymity network serves as a global infrastructure supporting cybercriminal activity on the Dark Web [11]. Tor Users can anonymously conduct transactions of illegal goods and services on marketplace websites supported by Tor OSes. Detecting sessions to these OSes is valuable for law enforcement agents, for instance to identify either offending clients or even the location of the OS itself.

While the desired features outlined above present significant advantages, they also introduce a range of

challenges that must be addressed to achieve an effective solution for the live correlation problem. One of the primary challenges is defining correlation queries in such a way that they yield results with minimal Response Time (RT). This requires a carefully designed query language that allows system operators to specify their needs clearly while ensuring that the system can process these queries efficiently.

Moreover, considering the system requirement to process large amounts of data at line speed, it also becomes necessary to create a distributed and scalable solution to accommodate large workloads and multiple queries from system operators and guaranteeing reliable end-to-end interaction between the distributed cluster nodes. Such requirements were achieved through an implemented combination of Apache Kafka [12] and Spark Structured Streaming [13] frameworks, working together to correlate data.

On top of this, the functionality of the entire developed design needs to be validated under a realistic environment, therefore the system will undergo extensive testing, considering every factor and the most relevant scenarios, to observe and analyze the behavior of the complete system.

1.2 Contributions

As a result of this thesis, we introduce CorrForce, along with a detailed description of its design, implementation, and evaluation as a distributed, live and query-based correlation framework. The achieved contributions are the following:

1. CorrForce, the first live flow correlation system aimed at identifying criminal and harmful activities over the web, including novel ISP collaboration mechanisms for traffic capture, and a distributed module responsible for performing correlation tasks;

2. The introduction of a new query language designed to accurately manipulate both temporal and spatial aspects of correlation, along with a web UI for simple interaction with the query system and easy visualization of results;

3. A prototype implementation over Kafka and Spark Structured Streaming, capable of executing correlation queries using the *SUMo* correlation model;

4. A review and performance optimization of *SUMo*'s correlation algorithm;

5. An extensive experimental evaluation of CorrForce's reliability, performance, and scalability was conducted across multiple scenarios, including live environments, reaching a preliminary throughput of 35 000 pairs/s, under a worst-case scenario and close to linear scalability.

1.3 Organization of the Document

The structure of the remaining chapters of the thesis are organized as follows. Chapter 2 summarizes the concepts and knowledge regarding correlation aspects and system specifications, necessary to follow the remaining chapters, while Chapter 3 presents the literature regarding the related work. Chapter 4 introduces the architecture and design choices for the proposed system architecture, and illustrates how CorrForce's operators can take advantage of queries for data correlation. Chapter 5 describes how we implemented a prototype of such design, while Chapter 6 presents and discusses all the performance results and observations for such prototype. Lastly, Chapter 7 gives a conclusion for the thesis and places some remarks for future work.

2

Background

Contents

2.1 Traffic Correlation Attacks 5

2.2 Other Correlation Use Cases 9

2.3 Desired Properties for Distributed Systems 11

This chapter presents an overview of the concepts and mechanisms that will be employed throughout this thesis. We begin with a general overview of traffic correlation in Section 2.1, followed the description of our case study, containing an overview of Tor's privacy service and how the such correlation techniques are applied to its context. In Section 2.2, we proceed to give a background over other possible use cases that can be explored by our system. Finally, Section 2.3 provides a perspective over the common properties associate with distributed systems, and how they conform with our system.

2.1 Traffic Correlation Attacks

Traffic correlation attacks aim at capturing segments of sessions, known as flows, usually gathered from multiple network sources, and pair them to determine the likelihood of both belonging to the same session. Several models

exploring this type of attack have been proposed, hence we will begin by examining some of this techniques and then explain how they are applied to the de-anonymization of Tor sessions.

2.1.1 Overview of Correlation Techniques

Flow Correlation techniques can be divided into two categories: passive and active. Passive correlation attacks comprise approaches that only observe on going traffic characteristics and derive the necessary features from the captured traffic. Meanwhile, active approaches, also named as flow watermarking, apply modifications to targeted traffic, embedded in channels such as content, timing, size or rate over the network protocol layer, in order to be able to perform correlation [14]. This thesis will focus on passive correlation attacks.

Early proposals for correlation attacks relied on statistical models, based on packet timing and sizes [15–17]. For instance, Shmatikov et al. [17] while proposing a new defense against timing analysis over small observation times, introduces a correlation model based on cross-correlation over entry and exit links extracted from HTTP trace based models, using packet counts from multiple small windows of time derived from the division of the observation window, and deeming the pair correlated if the obtained coefficient is larger than a threshold. These techniques were not very efficient due to the difficulty of the general-purpose statistical approaches to capture the network dynamics.

In recent research, correlation systems moderately rely on machine learning or deep learning approaches due to their ability to extract effective features from raw input data [18]. Nasr et al. [9] proposed *DeepCorr*, a pioneering deep learning algorithm able to learn correlation functions using the timings and sizes of bidirectional network flows. It employs a Convolutional neural network (CNN) that is trained on a large dataset of actually correlated flow pairs. By inputting the extracted characteristics (i.e. vectors of inter-packet delays and packet sizes) of a flow pair, the model returns a coefficient. If this coefficient surpasses an experimentally determined detection threshold, the flow pair is classified as correlated. When compared to previous techniques, *DeepCorr* achieved significantly better accuracy, being much faster in the correlation process for the same accuracy. However, *DeepCorr* still presented some performance limitations specifically the quadratic number of flow pairs it needs to correlate (i.e. if N is the number of current sessions, then it needs to correlate $N \times N$ pairs) and the high *False Positive Rate* (FPR) that negatively affects the success rate of the model. To counter these drawbacks, Oh et al. [19] presents *DeepCoFFEA* that similarly to *DeepCorr* relies on a deep-learning model based on packet timing and sizes but significantly reduces the cost for each comparison by embedding flows into a low-dimensional space and then using more efficient comparisons between these embeddings; and ii) reduces the FPR, without affecting the *True Positive Rate* (TPR), by employing an amplification technique to aggregate multiple comparisons from windows of the flow pair. Results were consistent across windows with a TPR at 97-98% for 12% FPR. After applying the voting strategy to aggregate the results, it achieved 97.99% TPR and 0.13% FPR, proving a significant drop in FPR as expected.

SUMo [2] is a novel correlation technique, that unlike *DeepCorr* and *DeepCoFFEA* does not rely on deep

learning algorithms. Rather, *SUMo* adopts a variation of an NP-complete decision problem, the subset sum, to perform flow correlation. The pipeline presented by this approach consists of two online phases, filtering and matching, and an offline training phase. The latter is utilized to determine optimal hyper-parameters for both the filtering and matching phases. The filtering phase selects only relevant flows, while the matching phase groups the select flows into time-overlapping pairs and applies a variation of the subset sum problem to sliding windows over the flow's packets. This results in a vector of scores that, when aggregated into a final score, are compared against an empirically determined threshold to validate the correlation of the pair.

SUMo has demonstrated impressive results, achieving over 99.6% precision and TPR when correlating sessions of any duration. While comparing it to the previous approach, given 5000 flow pairs as input, *SUMo* outperformed by yielding a peak throughput of approximately 153,000 pairs/s, while the *DeepCoFFEA* implementation exhibited 1639 flow pairs/s and worsening with larger input batches.

2.1.2 Tor Correlation Attacks

Tor is a well known low-latency service that provides anonymity to both ends of the communication [20]. By default, Tor achieves anonymity by routing the produced traffic through a pre-established routing circuit, formed by three relays: entry, middle and exit. Messages issued by a client are wrapped under multiple encryption layers (i.e. using TLS protocol) such that the intermediary relays cannot observe the contents of the message or learn the full route of the message within the relay network between sender and receiver. Using this mechanism, each relay in the circuit knows the relay that sent it the data and, by decrypting one of the encryption layers, it discovers the relay to which it is passing the data, hence the identity of the client remains concealed from any intermediate node aside from the entry relay [10, 21]. However, simply using this mechanism only provides anonymity to the client, and not to the receiver end of the connection, since the client must be aware of the address of the receiver. With this in mind, Tor employs a mechanism known as Onion Services (OSes). This technique utilizes a new type of address, that is advertised by the OS in multiple places across the web. Through this address and by making use of the aforementioned circuits, both the client and the OS can connect to an elected intermediate relay, named *Rendezvous Point*. Having two distinct circuits from the client and the OS to the rendezvous point, guarantees that both end points of the communication are anonymous [10].

The Tor anonymity service provides its users with the privacy required in various circumstances. This includes criminal practices, as Tor constitutes one of the pillars of the *Dark Web*, where many illicit activities such as marketing and promotion of pornography, drugs and weapons take place [11]. In these specific circumstances, the detection and de-anonymization would be desirable, hence multiple recent studies have identified potential attacks to which Tor is vulnerable [10, 21], that contribute to the achievement of these objectives. Among these attacks are the previously presented traffic correlation attacks, which are enabled by the low-latency nature of Tor, as hiding packets timing would make the service slow [22].

DeepCorr [9], *DeepCoFFEA* [19], and *SUMo* [2] were initially designed as correlation mechanisms for Tor flows.

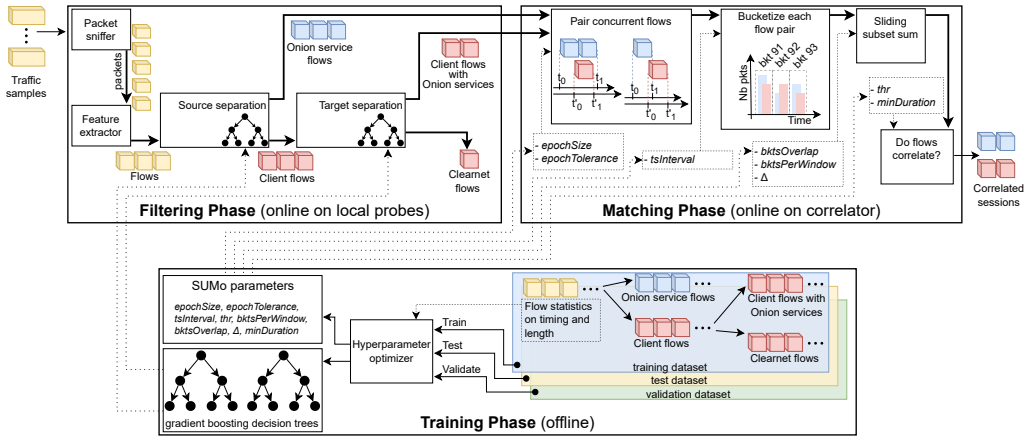


Figure 2.1: The SUMo pipeline comprising two online phases – filtering and matching phase – and one offline training phase.

In addition to the correlation performance discussed above, both *DeepCorr* and *DeepCoFFEA* primarily target Tor traffic to the clearnet, however, their performance in correlating flows to OSes remains unclear. Potential drops in performance may occur due to multiple reasons [2], for instance, differences in traffic characteristics such as manipulated packet sizes at the server endpoint, or the assumption taken by these approaches that a one-to-one correspondence between client-side and server-side flows is universal. OSes may connect with multiple clients concurrently, multiplexing traffic from various sessions through the same connection between the OS and its guard node, rendering the assumption inaccurate. Scalability bottlenecks may also arise as multiple combinations of pairs need to be tested, frequent model updates and data collection for training may be required as traffic patterns change and neglecting the low rate of the rare occurrences may lead to overestimation of the classifier's performance (e.g. classifying traffic to the clearnet as OS traffic, due to the lack of balance between the traffic volumes of each type) With all these factors in mind, *SUMo* will be the correlation mechanism that we will build the design of our system upon, hence we present in more detail its architecture.

2.1.3 *SUMo*'s Architecture

As discussed earlier, after being trained and having the optimal hyper-parameters, *SUMo* operates in two phases: the Filtering and the Matching. Figure 2.1 presents the complete *SUMo* pipeline, including the two phases we are describing.

Filtering phase: As a first step, *SUMo* discerns between client-side flows and OS-side flows, as clients tend to send smaller and receive larger volumes of traffic per flow when compared to OSes. Forwarding the former's results, the Client Flow (CF)s destined for OSes are isolated from those directed to the clearnet, using as base the difference between latency when accessing a page over the different network topology.

Matching phase: Given that comparing all CFs with every Onion Service Flow (OSF) would be inefficient, as the majority of flow pairs do not overlap temporally, *SUMo*'s algorithm only compares OSFs that temporally overlap

with the current CF, avoiding unnecessary processing. To compute the correlation between these temporally overlapping pairs, the algorithm employs the sliding subset sum approach. It first divides the packets of each flow in the pair into buckets (i.e. empirically determined coarser-grained time intervals), which aids in simplifying the algorithm's input and compensates for packet delays and reordering inherent in Tor's multi-hop network. Following this, for each bucketized pair, *SUMo* aggregates buckets in overlapping windows and calculates the subset sum pair similarity score per window, executing a variation of the subset sum problem. In this problem, a given a set of positive packet sizes, A, and a positive integer M representing total packets received by the client, the algorithm looks for a subset of A such that the sum of its members equals M. From this, the window progresses by a pre-determined offset, until the flow is fully processed. Upon the completion of the sliding subset sum computation for all potential flow pairs between the CF and its candidate OSFs, the algorithm generates a per window score vector that when averaged represents the final similarity score of the flow pair. It then selects the flow pair with the highest score among all pairs, under the idea that only one candidate can be correlated with the CF. Finally, the determined coefficient is compared against an empirical threshold to validate correlation, otherwise no candidate OSF is correlated with that particular CF.

2.2 Other Correlation Use Cases

As cyber threats become more sophisticated and the volume of data transmitted over networks continues to grow, leveraging advanced machine learning and data processing techniques in a scalable manner is imperative. This section delves into other use cases can make use of our framework to improve the fields of VPN traffic detection and classification, media piracy detection, and anomaly detection. Each use case presents unique challenges and solutions when deployed in a large-scale and real-time scenarios.

Detecting and classifying VPN traffic: Advancements in machine learning have enabled more effective analysis of VPN traffic, crucial for identifying a range of activities from benign data transmission to masked malicious activity. For this, Miller et al. [4] leveraged Pearson's Correlation Coefficient model to distinguish between VPN and non-VPN traffic, achieving an accuracy of over 92%. Bagui et al. [23] employed supervised machine learning models to detect and categorize VPN traffic into multiple categories like browsing, chat, file transfer, mail, peer-to-peer, streaming, and VoIP. They achieved the best results with Random Forest and Gradient Boosting Trees, demonstrating over 90% accuracy with a small number of time-related features, suggesting that these models can efficiently handle large amounts of traffic. Hartl et al. [3] presented a Recurrent Neural Network to separate individual traffic flows from VPN traffic from multiple applications being accessed simultaneously by the same user. This method leaves packet lengths and inter-arrival times observable, posing a risk for traffic analysis, but also being valuable for intrusion detection, and achieving an overall accuracy of over 94% in various scenarios.

Detecting media piracy: The proliferation of online platforms distributing copyrighted material illegally has led to the development of various detection mechanisms. For example, YouTube's signature-based detection [6]

has driven the development of evasion techniques, such as removing parts of the content and injecting artifacts to generate different signatures. Price [24] investigated the growth of illegal pirated content streaming, noting a 159% increase in bandwidth use from 2010 to 2012. Ibosiola et al. [25] studied the streaming cyberlocker ecosystem — services that allow users to stream potentially illegal pirated content — to find their characteristics, such as relationships between these websites. Rafique et al. [26] studied the ecosystem of free live streaming services, many of which serve illegal pirated content, and built a classifier to detect pages offering free live streaming services for further analysis on whether they are serving illegal pirated content. Rudman et al. [27] proposed a system for real-time detection of forensic watermarks in live streaming services. Stolikj et al. [5] used Convolutional Neural Networks to recognize logos to recognize the source of a video stream. The authors proposed it as a building block for a real-time illegal pirated content monitoring system.

Anomaly detection: Anomaly detection in network systems has emerged as a critical component in maintaining the integrity and performance of data centers and digital infrastructures. As networks become more complex and the volume of data they handle increases, traditional methods of monitoring and detecting anomalies become less effective. Anomalies in network traffic, which can range from unusual traffic patterns signaling a cyber-attack to subtle irregularities indicating system malfunctions, demand sophisticated detection mechanisms that can adapt and respond in real-time. The latest approaches include semi-supervised classifiers, streaming data processing algorithms, and advanced neural network architectures. They illustrate the innovative strategies being employed to not only detect anomalies more accurately but also to do so efficiently, with reduced computational overhead.

Decker et al. [28] developed a real-time anomaly detection system for data centers to produce multi-purpose logs using a semi-supervised classifier based on Gaussian membership functions. Habeeb et al. [1] addressed the limitations of state-of-the-art anomaly detection systems in real-time scenarios, and proposed a novel framework using big data technologies such as Kafka and Spark. Their framework utilizes a streaming sliding window local outlier factor coreset clustering algorithm, which combines several techniques that optimize large-scale data streaming in a real-time network context. The proposed algorithm achieved an accuracy of over 96% while reducing memory consumption and execution time when compared with other existing algorithms. Kim et al. [29] combined Convolutional Neural Networks and Long Short-Term Memory for intrusion detection, optimizing feature learning through normalized UTF-8 character encoding and employing Docker images for deployment. The framework performs continuous re-training to enhance the detection of unknown attacks.

Building on the foundation of advanced anomaly detection methods, another critical aspect of network security is the identification and mitigation of botnet threats. Botnets, networks of infected devices controlled by a central entity, or command-and-control server, pose significant risks, including DDoS attacks and unauthorized access to sensitive information. The following research contributions detail innovative methods developed to effectively detect and analyze botnet traffic. These methods utilize a variety of machine learning techniques and algorithms, showcasing the adaptability of these approaches in different network environments and against various botnet structures.

Khan et al. [8] produced a two-stage detection method to classify peer-to-peer botnet traffic. The first stage filters non peer-to-peer traffic, while the second stage detects botnet peer-to-peer traffic using a Decision Tree classifier. Ibrahim et al. [30] developed a two-stage framework based on a filtering phase that uses K-means clustering to differentiate normal and botnet traffic, and a classification phase based on a supervised machine learning classifier to detect the botnet's command and control server's IP. This framework targets multiple botnet structures, such as centralized and decentralized architectures. Kanehara et al. [31] applied tensor factorization and adaptive thresholding to detect botnets on darknet traffic and implemented Nonnegative Tucker Decompositions to extract features from input data in real-time. This way, the authors managed to identify the IP of collaborating malicious devices and their destination port numbers. Gromov et al. [32] proposed converting incoming traffic to images and feeding them to a Convolutional Neural Network for real-time detection of attacks by the Mirai botnet. The authors achieved a performance of 720 packets per second for an accuracy of over 99%.

The studies discussed, ranging from VPN traffic analysis to botnet detection, demonstrate significant progress. However, they also reveal gaps that future research must address. A key limitation is that many of these systems are not optimized for real-time processing scenarios. Even in cases where real-time capability is claimed, there is often a lack of rigorous testing under demanding conditions, such as managing the immense traffic volumes typical of an Internet Service Provider. Moreover, these systems frequently demonstrate a lack of versatility, limiting their applicability across different domains and use cases.

Our proposed system seeks to address these shortcomings by offering a flexible and scalable framework, specifically designed to adapt to a variety of scenarios. This framework is not only capable of handling high-volumes data efficiently but is also designed to be modular and extensible, enabling its application in diverse domains beyond its initial implementation. In doing so, it aims to set a new standard for adaptability and robustness in network security and traffic analysis tools.

2.3 Desired Properties for Distributed Systems

The wide range of existing distributed systems serves various contexts and operates under different environments. Despite this diversity, they share some common characteristics vital for their functionality. However, some topics such as fault tolerance, data privacy, and security are not the focus of this thesis and therefore were not addressed. In contrast, the relevant properties are the following:

- *High throughput*: Distributed systems should handle high volumes of data and process numerous records efficiently, achieving high throughput and responsiveness.
- *Scalability*: These systems must be scalable, meaning they can expand by adding more nodes or enhancing existing ones while ensuring that performance improvements are proportional to the resources added. Scalability is crucial for accommodating larger workloads and growing user demands without degradation the service quality.

- *Communication*: The distributed and concurrent nature of these systems requires robust communication between multiple components. Reliable and fast communication protocols are essential for maintaining consistency, coordination, and synchronization across the distributed network.
- *Declarative interactive language*: A declarative language allows users to interact with potentially large distributed systems in a simple and efficient manner. Facilitating querying and data manipulation.
- *Modularity*: These type of systems should be modular, enabling the integration of new features without overhauling the entire system architecture. This allows for incremental updates and adaptations, making the system more flexible and easier to maintain.

The desired system solution to be effective and efficient, must prioritize the above topics. For instance, high throughput and scalability, to accommodate large volumes of online data, reliable communication and declarative languages, for a simple interactions and good user experience, and modularity, to easily integrate new features(e.g. new correlation use cases aside from *SUMo*).

Summary

This chapter delves into traffic correlation attacks, exploring various correlation techniques, including statistical methods and machine learning algorithms and highlighting their role in analyzing traffic patterns to correlate data flows across a network. Special attention is given to Tor correlation attacks, illustrating the vulnerabilities within the Tor network. In particular, it explores the architecture of *SUMo*, the current state of the art Tor traffic correlation model. Then, details regarding other crucial traffic correlation use cases are discussed in, namely detecting VPN traffic, media piracy, network anomalies. The chapter concludes by outlining the desired properties for distributed systems, such as high throughput, scalability, reliable communication, a declarative interactive language, and modularity, which are essential for solution developed in this thesis. In the chapter that follows, we survey related research with relevant efforts aligning with ours.

3

Related Work

Contents

3.1 Big Data Processing Technologies and Techniques 14

3.2 Distributed Query Processing 16

3.3 Distributed Machine Learning 18

3.4 Traffic Analysis and Networking 19

3.5 Real-Time System Metrics 20

One of the big goals of this research is to achieve a system design that can efficiently process large volumes of data, thus we begin this section by presenting the most widely used big data processing approaches, as they are often able to tolerate varying workloads even under unfavorable circumstances. Then we introduce systems that employ distributed query processing methodologies and optimization techniques, to inspire the query interpretation mechanisms on our proposal. As our study case relates to traffic analysis and ML-based correlation mechanisms, we cover related large scale systems that perform it with distributed real-time approaches. Finally, we give insights on how some real-time systems can have their performance evaluated.

3.1 Big Data Processing Technologies and Techniques

Dealing with large sets of data requires the usage of efficient models that perform a quick analysis and response to dynamic data streams. Designing such models however is challenging, given the scale required for the largest applications, which can involve hundreds of nodes. Big data processing models have evolved to handle these complex data volumes, and effectively address challenges regarding resource managing, work scheduling, fault tolerance and straggler detection [33]. As in many systems described ahead, our correlation approach will need to either employ these technologies, or use similar techniques to address the aforementioned challenges and process the correlation tasks over continuously growing flow sets from the ISPs. Therefore we present in detail the most used technologies for this purpose, separating them into the two types of processing approach, batch and stream processing.

3.1.1 Distributed Batch Processing

The batch processing paradigm is a technique where large static datasets are processed in batches rather than in full, without the need for intermediate user intervention, optimizing efficiency concerning the voluminous data.

The pioneer MapReduce, originally proposed by Google [34], is a fault-tolerant batch processing framework designed to handle extensive datasets in parallel across numerous low-end computing nodes. Despite the rise of recent technologies and methodologies for big data processing, MapReduce remains in use in specific environments due to its reliability and scalability when dealing with vast amounts of data. Such data is partitioned across multiple machines in a Distributed File System (DFS) and represented as (key, value) pairs. Map tasks receive the key-value pairs as input, producing new ones as result. These are then grouped and passed to Reduce functions assigned for each key, to produce the final output. Apache Hadoop[1] is an open-source tool comprising an implementation of the MapReduce framework, along with other big data systems, such as the Hadoop Distributed File System (HDFS).

3.1.2 Distributed Stream Processing Engines

Unlike batch processing, which handles static data, distributed stream processing engines manage vast amounts of continuous data in real-time, providing scalability, fault tolerance, and high throughput [35]. This thesis surveys some of the most relevant systems available today.

Spark Streaming [36]: It is a stream processing extension to Apache Spark that uses discretized streams (D-Streams) for structuring streaming computations as stateless, deterministic micro-batch computations. Its architecture includes a master node for coordination, multiple worker nodes for data partition storage and task execution, and a client library for data input. Spark Streaming maintains state through Resilient Distributed Dataset (RDD)s, fault-tolerant data structures that keep data in memory and allow deterministic recomputation

[1] https://hadoop.apache.org/

by tracking the lineage graph of operations. Node failures are managed by recomputing RDD partitions using parallel recovery and periodic checkpointing to prevent infinite recomputation. Straggler mitigation is achieved through speculative execution of slow tasks. Master recovery is supported by reliably writing the computation state at each time-step and having workers report RDD partitions to a new master in case of failure.

Spark Structured Streaming [13]: Enhances Spark Streaming by supporting both D-Streams micro-batch mode and a new continuous processing mode, which offers lower end-to-end latency at the cost of less operational flexibility. A new query model is employed to automatically analyze and optimize queries, making the system easier to use. State tracking is managed through two external storage systems: a write-ahead log for durable and atomic writes at low latency, and a state store for storing larger amounts of data durably with parallel access. These systems enable recovery from failures in both modes. However, only the micro-batch mode can recover from stragglers and address load imbalances, leveraging its small task granularity.

Apache Flink [37]: Apache Flink supports both batch and stream data processing, providing distinct APIs for each type. Regardless of the API, all Flink programs compile down to a common representation: the *Dataflow Graph*, which resembles a *Directed Acyclic Graph* and can be associated with one or more sources and sinks. These graphs are composed of data streams (originating from message queues, socket streams, file sources, or by transforming other data streams) and operators that apply transformations to data streams. Each data stream is divided into multiple partitions, while operators have one or more sub-tasks to enable parallel and distributed processing. Flink's execution pipeline includes a client that transforms program code into a dataflow graph, which is then submitted to a Job Manager for coordinated distributed execution. Task Managers process and materialize data streams, manage multiple operators, and report their status to the Job Manager. For fault tolerance, Flink employs a combination of stream replay and checkpoint mechanisms, allowing the system to restore the state of operators and replay events from the checkpoint onward.

Apache Kafka Streams[2] **[12]:** Kafka Streams, a component of the complete Apache Kafka system following the central abstraction of unbounded and immutable data records represented as a stream. The computational logic for stream processing is defined through a graph composed of stream processors connected by streams (edges) or shared state stores. Special processors, namely source and sink, facilitate the input and output of the topology. To define this topology, Kafka Streams provides two methods: the Kafka Streams Domain Specific Language, offering common data transformation operations, and the Processor API, for custom processor definitions. The system partitions data for processing, utilizing stream partitions and stream tasks as logical units for parallelism. State stores support storing and querying data in order to achieve stateful operations. Kafka Streams ensures fault tolerance by leveraging Kafka's highly available and replicated partitions, for instance, in case of task or machine failure, it automatically restarts tasks on remaining instances of the application.

Other Engines: IBM Streams [35, 38] stands as a highly customizable stream processing engine equipped with multiple toolkits (e.g. for development environment or analytics), data partition methods (e.g round-robin, ran-

[2] https://kafka.apache.org/documentation/streams/

dom, or hash) and fault-tolerance mechanisms. In the context of IBM Streams, an application is represented as a directed flow graph where nodes represent stateful operators. Apache Storm's[3] flow can be characterized by spouts, bolts and tuples (i.e. stream sources, processing nodes, and intermediate results). Spouts generate tuples that are processed by bolts resulting in new transformed tuples. Furthermore, tuples can be partitioned across bolt's tasks without constraints. LogFlash [39] was proposed as a log-based real-time anomaly detection system on software. The system was shown to be more effective than previous systems at handling real-time stream processing using Hadoop, Apache Spark, and Apache Flink. Cameo [40] is a fine-grained multi-tenant stream processing system designed to withstand workload variability that endangers the performance and efficient resource utilization of slot-based approaches like Apache Spark, and Apache Flink, where operators are assigned dedicated resources. However, previous fine-grained approaches like Orleans can suffer from high tail latencies. Cameo achieved lower latency and less overhead than Orleans. Railgun [41] is a stream engine aimed at detecting financial fraud. It operates using real-time sliding windows to ensure continuous aggregation correctness, rather than only at specific intervals. Railgun lacks stream-stream join capabilities and primarily focuses on data aggregations over extended periods of generated data. However, it leverages other technologies such as Kafka, to facilitate distributed processing, fault-tolerance and recovery, and RocksDB to maintain aggregation states.

Overall, based of the analysis performed in [35,42], Spark and Storm stand out among other stream processing engines due to their maturity and larger user base, extensive training materials, and support for third-party libraries. Isah et al. [35], also highlights that when requiring complex stream processing and machine learning analytics, Spark and Flink will be the best options because of their built-in libraries for large-scale computations and their support for machine learning algorithms. For time critical applications that require a collection of services and hard real-time processing, Kafka Streams and IBM Streams are seen as good options.

3.2 Distributed Query Processing

Some query types can often involve complex operations in database systems, such as joins, aggregations, and sub-queries [43]. Traditional databases may struggle to process such queries with exceptionally high workloads [44]. On top of this, query complexity can grow depending on the context they are inserted, for instance, being simply related to large scale systems, or due the sizable combinatorial nature of the problem, similar to the one in CorrForce. Hence, new distributed query models emerge, often involving optimizations in the query execution planning, as the demand for low latencies, typically in the order of seconds, requires query interpreters to generate high-quality distributed execution plans.

Sun et al. [44] work illustrates the large combinatory problem of computing data similarity, developing Dima, a distributed similarity-based query processing system over SQL, supporting multiple data operations (i.e. similarity selection, similarity join, top-k selection and top-k join). This system was incorporated into Spark, relaying on its query optimization techniques. Furthermore, to support such operations while avoiding expensive data

[3] https://storm.apache.org/

transmission in distributed environments, Dima utilizes balance-aware signatures (i.e. two records are similar if they share common signatures), so that it can adaptively select the signatures to balance the workload.

Motivated by the inefficiency of previous methods for sensory data query processing in traditional wireless sensor networks (WSNs), Cai et al. [45] researched the effectiveness of query processing in an edge-assisted Internet-of-Things (IoT) data monitoring system (EDMS). Opposing to those methods, edge servers in an EDMS provide computation and storage resources to nearby IoT devices, offering faster data processing. In combination with EDMS distributed approach, queries over massive volumes of sensory data can be processed, without putting all the stress on the cloud. This article introduces a performance bounded algorithm for the NP-Hard distributed query processing latency minimization (MIN-QPL) problem, under EDMS.

Query optimization: Query optimization is one of the most challenging problems in database systems, with the common goal of achieving an efficient query execution plan, often requiring complex hand-tuning. Achieving such optimal plan can not only improve the execution time of the query but also aid in the resource management problem, allowing for an efficient utilization, therefore we give particular emphasis to this topic, as it will give favorable insights to use in our approach.

Kossmann et al. [46] provides an overview, including description and classifications of existent query optimization techniques, along with execution strategies that are enabled by data dependencies. It outlines three central activities that are usually combined in the query optimization process, in the majority of the systems, consisting in: i) Cost-independent transformation: describes the process of rewriting a query into a equivalent but rather efficient version through static, rule-based transformations (e.g. in SQL, to perform joins we can either use the *WHERE* or the *JOIN* clause to formulate the query, the latter is more efficient) ; ii) Cardinality and cost estimation: focuses on estimating the cost of a query plan a priori; iii) Cost-based transformation: aims at enhancing physical query plans via cost-driven transformations that depend on the specific database instance.

Recent optimization techniques are often based on learning approachs. For instance, Marcus et al. [47] introduced Neo, a query optimizer based on deep neural networks to generate query executions plans. It builds over other optimizers and gains knowledge from incoming queries, upon its successes and failures, allowing it to adapt to underlying data patterns and enduring estimation errors. Similarly, Bao [48] is a query optimization mechanism based on tree convolutional neural networks. It analysed the issues from previous approaches and leverages existing query optimizers (e.g. PostgreSQL optimizer) to provide per-query hints in the optimization process. Through the generated hints, Bao is able to quickly learn from mistakes and adapt to changes in query workloads, data, and schema, even under tail latency (i.e. learning techniques are better on average, but may perform terribly in some situations), while improving end-to-end query performance. A comparison with previous approaches, including Neo, shows that it significantly stands out when under dynamic workloads.

Finally, we can deduce from the above work that for an efficient distributed query execution module it is often required both rigorous query execution planning and the application of optimizations over how and what data should be processed, enabling large queries to finish within reasonable timing.

3.3 Distributed Machine Learning

In the dynamic field of data science, developing distributed learning approaches for online processing of large-scale data has become increasingly crucial [49]. This includes addressing storage and bandwidth challenges in handling big data datasets. Recent advances in machine learning have also brought real-time processing needs for applications such as immediate decision-making in autonomous vehicles and enhancing user experiences with real-time personalized responses.

Recent studies have focused on leveraging big data technologies and machine learning algorithms to enhance the efficiency and effectiveness of real-time data processing and integration with training pipelines. Some correlation mechanisms discussed earlier utilize ML-based approaches, and we intend to explore how these systems employ existing distributed real-time techniques to enhance their performance.

Chebbi et al. [50] utilize a master-slave architecture based on HDFS and Apache Spark for storing and processing large volumes of satellite images, integrating TensorFlow for feature extraction and classification. However, real-time metrics were not explicitly discussed. Syed et al. [51] survey big data technologies in smart grids, focusing on large data volume management and analysis using machine learning. They review three major big data architectures: Hadoop for offline batch processing, Apache Storm for real-time stream processing, and Apache Spark for both. They suggest installing Apache Spark atop Hadoop to leverage Spark's analytical capabilities alongside Hadoop's efficient data storage system. Tang et al. [52] review the Apache Spark ecosystem, highlighting its effectiveness in optimizing real-time machine learning applications using tools like SparkNet. Spark is currently used in genomics [53], healthcare [54], finance [55], and astronomy [52]. These works often rely on big data technologies for their simplicity and effectiveness.

Federated learning: Federated learning (FL) surged to prevent the need for users to share their sensitive information. This way, each participant can collaboratively train models with their local data, allowing for distribution and acceleration of time consuming tasks associated to training a model. SmartPC [56] is an hierarchical online pace control framework for FL aimed at optimizing the balance between training time and model accuracy with energy efficiency, presenting 32.8% reduction in energy consumption, while speeding up the training time by 2×. A global controller assesses the status of each participating device and selects qualified ones, assigning them estimated deadlines to ensure adaptive progress in the training. Each device dynamically adjusts its setting through a local pace controller in order to meet the deadline, while reducing power consumption. Zhang et al. [57] proposed an asynchronous FL approach applied to the automotive industry for steering wheel angle prediction in autonomous vehicles. This approach improves the local model performance by using an asynchronous aggregation protocol and evaluated it based on model training time and bandwidth cost. This system reduced training time by 70% and bandwidth cost by 60%. Lastly, despite these systems being more closely related to network edge computing than to traditional distributed systems, they still carry important insights that we can use when choosing the distributed real-time design for CorrForce.

3.4 Traffic Analysis and Networking

The major challenge to traffic analysis [58] is scaling to today's gigantic volumes of network traffic. There is a variety of systems that already propose distributed real-time services. However, most of these services employ domain-specific optimizations. Notwithstanding, real-time systems have substantial and varied applications [7] in the networking domain, such as web analytics, advertising, media, edge computing, and biometric devices. Some of this analysis systems were already introduced as possible use cases for future support on CorrForce, thus we aim at understanding how previous work on this topic achieve real-time processing capabilities.

Anomaly detection: Detecting malfunctions and cyber attacks in real-time is crucial to prevent dangerous consequences. Several real-time systems have been proposed for anomaly detection [7,28]. Habeeb et al. [7] reviewed existing real-time systems, highlighting challenges such as redundancy, computational cost, data structure, noisy and missing values, hyperparameter tuning, inadequate architectures, difficult visualizations, data heterogeneity, and lower accuracy. Decker et al. [28] developed a log-based real-time anomaly monitoring system using semi-supervised learning with Gaussian membership functions, but it is limited to log-based systems. Kim et al. [28] implemented an intrusion detection system based on an optimized CNN-LSTM architecture for real-time processing, though it is domain-dependent and lacks real-time metrics validation. Habeeb et al. [1] proposed a clustering-based anomaly detection algorithm using Apache Kafka, but it remains domain-specific and its evaluation focused on execution time and memory consumption compared to previous algorithms.

Detecting botnets: There has also been increased interest in developing real-time systems specific to a type of cyberattack, such as botnets [31], due to their stealthy nature and destructive potential. Kanehara et al. [31] applied Nonnegative Tucker Decomposition, but these optimizations are domain-specific and won't necessarily generalize or would require specific adaptation to the domain.

Twitter spam detection use case: X (formerly Twitter) faces significant challenges with misuse, including product advertisements, phishing, and malicious links. Protecting legitimate users from such spam is crucial. Given the platform's immense scale, with millions of tweets daily, an efficient real-time system is essential.

This challenge has been a focal point of research, with numerous scholars proposing various supervised machine learning strategies [59, 60]. Twitter combats spam using a blend of manual reporting and automatic detection algorithms that identify suspicious behaviors based on graph-based features typical of legitimate users [61].

A notable advancement is the real-time spam detection system proposed by Sedhai et al. [62]. Utilizing semi-supervised learning, this system was evaluated using a dataset of at least 35,000 tweets per day over 15 days, showing an average processing time of just 0.495 ms per tweet. However, this approach has limitations, including domain-specific constraints and the lack of real-world, real-time testing with comprehensive metrics.

In summary, despite the numerous real-time systems in traffic analysis and networking, they often lack generalizability to other domains and are evaluated with metrics that reflect real-time performance rather than merely being faster than the state-of-the-art.

3.5 Real-Time System Metrics

Nowadays, evaluating real-time systems is increasingly critical. These systems, essential for applications ranging from autonomous vehicles to anomaly detection, require robust performance metrics. Key metrics such as latency, throughput, scalability, and fault tolerance ensure these systems can process and respond to data swiftly and efficiently. As data volume and complexity surge, these metrics are crucial for maintaining the integrity and effectiveness of real-time systems, ensuring accurate and timely results in an evolving digital landscape. However, these metrics may not reflect a fundamental characteristic of CorrForce: the worst case query response time. Hence, let us now discuss another fundamental metric often overlooked in the literature.

Worst case response time: Analyzing this metric plays a key role in evaluating the usability of a system in a real-time scenario. However, estimating the worst-case scenarios accurately is complex due to the variability in system behavior and the multitude of factors influencing performance, such as network latency. In our case, this metric would translate into the maximum time a query can take to return a result. The first proposed systems [63, 64] relied on static analysis, which is the process of analyzing the source code of a program without executing it. However, they may fail to consider all the factors that influence execution time, such as the expected execution path of the software in question (some functions will be executed more times than others, while some blocks might never be reached, for instance). Other hybrid analysis approaches combined static analysis with probabilistic analysis (branch prediction) [65], genetic algorithms [66] and search algorithms [67], but they pose restrictions on the software's code or lack explainability. Bate et al. [68] further employed search algorithms while ensuring the reliability of the measurements by bounding the variability and making statistical assessments of convergence of the execution times. However, the authors concluded that more work still needs to be done to complement search algorithms to aim to find the longest possible path instead of prioritizing only coverage. Búr et al. [69] introduced a semantic-aware worst-case execution time monitor using graph models to estimate the which path will take the most time to complete. The authors also evaluated the system on real-time query-based programs, finding that, the worst-case execution time was sampled in most testing cases.

Summary

In this chapter, we review related work relevant to our research. We start by exploring big data processing technologies and techniques, where we present the most relevant, distributed batch processing and stream processing engines. Then we delve into distributed query processing, highlighting their significance in managing large quantities of data. We also discuss distributed machine learning, as well as, traffic analysis and networking technologies. Lastly, this chapter tackles the commonly used real-time systems metrics. The next chapter will present the architecture for the proposed system, CorrForce, and introduce the query language we have developed to facilitate correlation for CorrForce's users.

4

Design

Contents

4.1 CorrForce Architecture Overview 22

4.2 ISP Probe 23

4.3 Correlator 28

4.4 User API 31

4.5 CorrForce Limitations 35

With the main goal of performing large-scale traffic correlation, we introduce CorrForce. Section 4.1 provides an overview of CorrForce's architecture and illustrates its placement within the ISP's network, enabling live correlation attacks. This is followed by a detailed description of each pipeline stage: Section 4.2 addresses how ISPs integrate the pipeline, including the capture and pre-processing of flows, while Section 4.3 details the application of live correlation to these flows, breaking down and reviewing all steps and requirements. Finally, Section 4.4 explains how queries are defined and utilized to leverage the available correlation mechanisms.

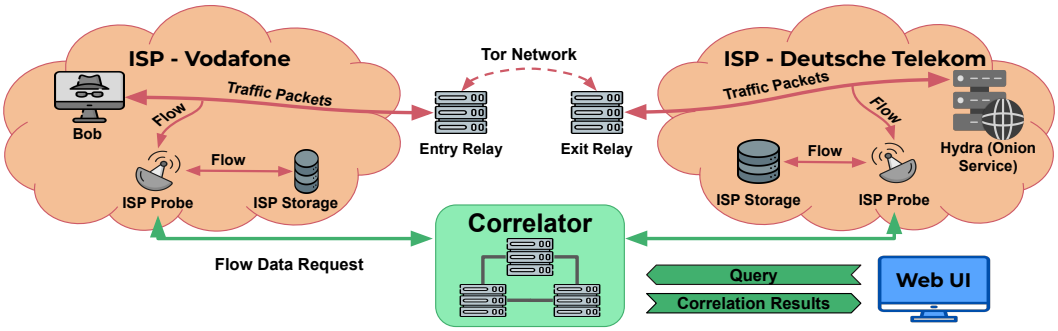


Figure 4.1: Architecture overview.

4.1 CorrForce Architecture Overview

By leveraging techniques from recent advancements in correlation attacks [2, 9, 19], CorrForce emerges as a system that empowers Law Enforcement Agenciess (LEAs) to effortlessly identify recent sessions of the Tor network between collaborating ISPs. This capability can be employed in uncovering criminal activity.

Figure 4.1 depicts a possible use case for CorrForce, using *SUMo* [2] as the correlation model. In this scenario, a fictional Tor user named Bob accesses an OS called *Hydra*, which was classified as the largest Dark Web Marketplace until its shutdown in April 2022 [70, 71]. Assume that Bob uses his personal device.

In this example, *Vodafone* (a Portuguese ISP) handles Bob's traffic while *Hydra* is located in *Deutsche Telekom* (a German ISP). ISPs directly manage the traffic exchanged by their customers, giving them the ability to observe interactions (i.e. sessions or flows) with Tor relays. Our system enhances the ISP capabilities with a component that filters and pre-processes such traffic, namely, the ISP Probe. This traffic is then prepared for the subsequent correlation phase. To maintain the identified flows, we also introduce an ISP storage module that acts as a cache, accessible by the Correlator later on.

LEAs that need to analyze traffic do so within a framework where ISPs consent to CorrForce accessing their traffic for the necessary analysis. The typical CorrForce user is therefore an LEA officer actively working to unveil suspicious Tor sessions occurring through the ISPs.

With this objective in mind, the LEA officer performs queries through the CorrForce *Web UI*, instructing it to correlate the data obtained from the ISPs in question. Within the Correlator, as its name suggests, the pre-processed data obtained from the ISPs undergoes correlation to de-anonymize sessions. If both ISPs contain sessions that are concurrent (i.e. temporally coincident), they are potential candidates for correlation.

In the example from Figure 4.1, *Vodafone* has a client session (CF) from Bob accessing an OS, and *Deutsche Telekom* has an OS session (OSF) from the *Hydra* OS serving a client. In this case, there is a high chance that the Correlator will flag these two sessions as correlated.

As we present on this example, CorrForce is built on three central components: *i*) the ISP Probe; *ii*) the Correlator; and *iii*) the user interface (*Web UI*).

ISP Probe: Each ISP is responsible for managing its clients' exchanged traffic, giving it the ability to capture such traffic. Our approach focuses on enhancing the ISP with custom filtering and traffic preprocessing capabilities to form flows and extract the necessary features for the supported Correlator mechanisms to uncover ongoing anonymous sessions. This component is implemented by the ISP to account for such features. Section 4.2 provides more details on how the ISP should proceed, including its architecture and how it interacts with the remaining components and ISP modules.

Correlator: The Correlator is the major component of the system, responsible for de-anonymizing sessions. This component has three main tasks: *i*) receive queries from a user, interpret their content to correctly execute the requested correlation, and return the results to the user who issued the query; *ii*) gather the pre-processed flow features from the specified ISPs; *iii*) perform the correlation over the gathered data using distributed versions of the supported correlation mechanisms. As of the current CorrForce development stage, the *SUMo* approach described in Section 2.1.3 is the only supported correlation mechanism, and it will be used to demonstrate CorrForce's utility throughout the thesis. Section 4.3 will provide a more thorough description of the challenges faced and elucidate how the final Correlator architecture was conceived.

User API: The interface through which the CorrForce user performs actions. A *Web UI* is provided to simplify the interaction with CorrForce. Its primary objective is to allow users to specify queries with a high degree of customization, enabling the highest accuracy and relevance of the obtained results. A secondary objective is to display results and general information, encompassing the following features: *i*) display the obtained results for a given query in the form of correlated flow pairs, whether the query is recent or older (i.e. recently issued queries with results stored on the user's local device or results from past queries kept by CorrForce itself); *ii*) for each resulting pair, access the metadata of each flow (e.g. the number of packets, each packet's size, and timestamps); *iii*) other secondary features, such as visualizing the identified flows in specific ISPs and presenting statistics related to query execution performance, among others that may be relevant in future work.

4.2 ISP Probe

The first step in CorrForce's pipeline consists in collecting the relevant flows within the network of the various collaborating ISP to later be fetched by the Correlator. Most ISPs integrate methods to collect and analyze the traffic from their clients, for instance to understand customer service, usage patterns or even to detect possible ongoing attacks on the their networks [72].

Therefore in this section, we introduce the ISP Probe and all the details for how an ISP can integrate it to collect flows. Then, considering the *SUMo* use case that we have been following, we introduce a component capable of aggregating Tor traffic into flows and classifying them into the respective types, i.e. CF or OSF, as mentioned in Section 4.1. Figure 4.2 represents the complete pipeline of the ISP Probe and provides a reference for the placement of its modules within the ISP.

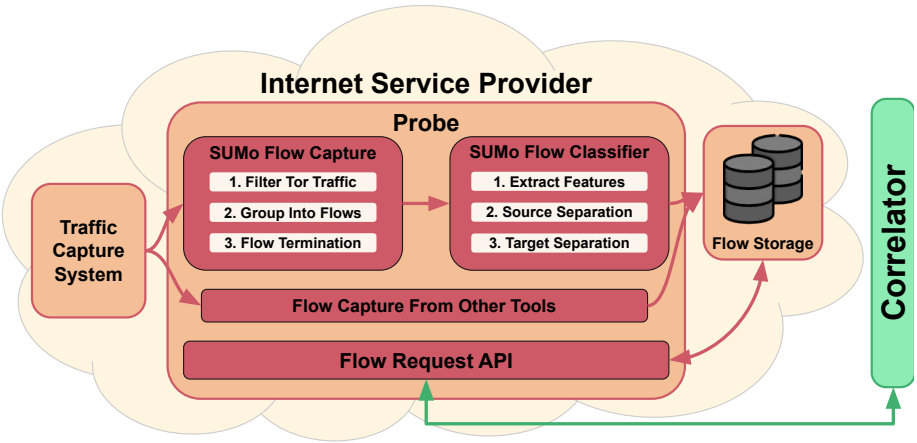


Figure 4.2: ISP probe flow capture pipeline.

4.2.1 Probing Policy

Integrating an ISP into CorrForce's range of action requires an agreement with the ISP management unit regarding the probe features (e.g. what traffic to capture, which correlation tools to support, etc.) to be integrated. This agreement, referred to as the Probing Policy, outlines all necessary information and configurations indispensable to both the ISP in question and the Correlator, with copies available within both, ensuring seamless access to ISP information across the system. The format of the Probing Policy file is presented in Table 4.1.

The **Probe Application Manifest** section comprises all the attributes regulating the collected and analyzed data for the flow aggregation and query progression. Among such attributes are the name identifying the ISP and a contract accorded with the ISP representatives legally authorizing all the behavior of the probe mechanism to be integrated and validating all the remaining attributes of the policy, paired with a creation and expiration date bounding the validity of the agreement. Most of the remaining parameters declare the range of action for the probe. Traffic data limitations, impedes any undesired traffic data from reaching the probe (e.g. traffic content, encrypted or not, due to legal and privacy matters). IP subnets and geographical regions define what traffic the ISP manages that CorrForce can access (e.g. ISP_X controls traffic from China, however due to the heavy privacy constraints in China, the country may not be listed in the regions). Maximum flow size and storage period avoid overflowing storage with possibly useless flow data. The list of supported correlation models and respective flow descriptors describe which flow data to collect and how it will be structured in storage. The last attribute defines the point of access for the Correlator to connect with the ISP to fetch the necessary flows for the ongoing queries.

The **Resource Configuration** section reflects all the resources to be allocated onto the probe mechanism employed, namely the number o virtual CPU units and amount of RAM memory, for the required efficiency level for the flow aggregation and classification, and disk size for the persistent flow storage.

Table 4.1: Probing Policy parameters.

Parameter	Description	Example
	Probe Application Manifest	
Name	Name of the ISP.	Vodafone
Creation date	Date of the agreement establishment.	01/01/2024
Expiration date	Expiration date o the agreement, requiring renovation of the contract.	31/12/2024
Contract	Valid contract established with the ISP management unit and proper authorization documentation.	-
Traffic Data Limitations	Matching condition filtering out unauthorized traffic data from entering the probe pipeline.	Packet content data excluded.
IP Subnets	List of IP address subnets belonging to the ISP in question that can partake in the flow traffic analysis.	[123.456.0.0/16, 222.333.0.0/16]
Activity Regions	Regions supervised by the ISP and authorized for flow traffic analysis.	[Germany, Austria, ...]
Maximum storage period	Maximum time a flow is kept in storage.	432 000 seconds (5 days)
Maximum flow size	Maximum size (in Bytes) allowed for a single flow.	1 KB
Supported correlation tools	List of supported correlation mechanisms and respective flow descriptor specifying the necessary flow features to collect.	*SUMo* and respective client/OS flow descriptor.
Flow Request (FR) API URL	URL or IP address/port defined for the FR API.	123.456.789.100:8080
	Resource Configuration	
vCPU cores	Number of vCPU core units to be deployed by the the ISP for the probe mechanism.	3 vCPU
Memory size	RAM memory allocated for the probe mechanism.	8GB
Storage size	Amount of disk storage (in GB) to be allocated for flow storage.	200GB

4.2.2 Flow Management

With the possibility of multiple correlation tools at hand, the ISP may need to manage not only large volumes of flows but also a great variety of flow types. The effective flow management requirement hinges on addressing three critical aspects, namely, how flows are defined, stored, and efficient loaded from the ISP.

Flow Descriptor: When supporting more then one correlation tool behavior, the ISP needs to be aware of the proper flow structure particular to each correaltion tool, ensuring that the correlation of said flows proceeds as expected. For this purpose, the *Supported correlation tools* of the Probing Policy carries the list of supported correlation mechanisms and respective flow descriptor. The flow descriptor defines the correct format to follow when creating a flow, containing all the features required for correlation (e.g. list of packet's timestamps and sizes, average size, start and finish timestamp, protocol, ports, etc.). A descriptor's set of features varies according to needs of the respective tool. Table 4.2 displays the features employed in the particular case of *SUMo*, the only currently integrated correlation mechanism.

Flow Storage: Upon the conclusion of capturing and classifying flows, they will not be immediately consumed by the Correlator, creating the necessity of a storage module to cache such flows within the ISP. In our point of view, we describe it as being a limited size storage (i.e. similar to a circular buffer) containing only the most recent identified Flows. The limited dimensions of this storage module is attributed to the expectancy that large volumes of flows will constantly be generated, hence if this storage module is too large it will become expensive to manage. As such, it should only keep a certain amount of flows and when necessary, older flows shall be

Table 4.2: *SUMo*'s flow descriptor features.

Feature	Description	Size (Bytes)	Example
Flow ID	Unique id to identify the flow, derived from the the IP addresses and respective ports of the communicating flow parties.	43	123.456.100.200:43899-700.273.500.300:35949
Query ID	ID associated used by the correlation tool to identify the which query it is referent to. Kept empty while stored locally and updated upon being fetch by the Correlator.	32	$sha256(ID_{User} + date)$
Flow Type	Flow type classified as either CF or OSF.	4	1 for CF and 2 for OSF
Initial packet timestamp	Timestamp in seconds of the first packet received.	4	1734304170.213 seconds (15/12/2024 23:09:30.213)
Final packet timestamp	Timestamp in seconds of the last packet received.	4	1734314170.672 seconds (16/12/2024 01:56:10.672)
List of packet timestamps	List containing timestamps of the N captured packets. In case of CF type, the list refers to the incoming packets (i.e. packets originated in the Tor Relay with client as destination). In contrast, the list in OSF type refers to outgoing packets (i.e. from the OS to the Tor Relay).	$N \times 4$	[1734304170.213, ..., 1734314170.672]

deleted. However, we do not place any hard constraints on how the storage module should be designed. Since the ISP manages this module, we allow a large degree of customization to improve flexibility in integration. The Probing Policy includes three configurations, i.e. *Storage size*, *Maximum storage period* and *Maximum flow size*, to limit the amount of storage used.

FR Handling: The final aspect involves accessing the ISP to retrieve the flow data. The Correlator can efficiently accomplish this by submitting a FR to the FR API. Through this request, the Correlator specifies the correlation mechanism to be employed along with other constraints to be applied over the selection of flows. In Section 4.3, when introducing the Correlator's query system, we will see in detail how the interaction between the Correlator and this module will happen, and how the to properly define a FR for the desired query outcome.

4.2.3 Use Case: Probing pipeline over *SUMo*

With *SUMo* serving as the current correlation use case, we illustrate the probing pipeline from its perspective, detailing each procedural step as depicted in Figure 4.2, including the filtering, grouping, and classification of Tor flows. Setting off from the assumption that every ISP already has a traffic capture mechanism in place, the volume of traffic it collects will undoubtedly consist of a large number of packets per unit of time. From such collection of traffic, only a fraction corresponds to traffic related with the Tor network and from this fraction only a small portion is relevant to form the required Tor flows. Therefore, a filtration stage over the gathered data is important for the success of the subsequent stages. The filtration is composed of two conditions: i) TCP protocol; ii) Packets exiting the Tor network.

TCP protocol packets: All communication to, between, and from Tor relays is done with TLS over TCP, hence all non TCP traffic is filtered out. On top of this, we do not make any assumptions on the ports used as both Tor Relays and Tor Clients can select a random port for the connection.

Packets exiting the Tor network: The next step is to filter non-Tor traffic packets. We use the list of currently

active Tor relays[1], updated every 30 minutes. If either the source or destination IP address from a packet's metadata matches a listed Tor relay, the packet is identified as Tor traffic. However, we are only interested in traffic entering or leaving the Tor network. Therefore, only packets with exactly one IP (either source or destination) matching a Tor relay will be considered to form a flow.

Having the relevant traffic, the next step is grouping the packets into flows. A flow essentially corresponds to a complete or partial TCP stream initiated by either a client or an OS with a Tor relay, and therefore it can easily be indexed by its source and destination IP address and port, coupling together all packets with such matching metadata. Detecting the end of a flow is comparable to the TCP stream ending procedure. The first goal is to capture a packet containing the FIN flag in its metadata, as TCP streams finish by exchanging FIN, ACK packets between parties. Secondly, TCP streams can suffer from large delays or failures in the communication, leading to FIN packets never arriving. Thus we set a period where packets may not arrive for a particular flow before raising a timeout. Unrelated to the TCP stream mechanics, flows are capped to the maximum number of packets as per the parameters defined in the probing policy. Notice that capping flows is not equivalent to discarding the remaining portion of the flow (i.e. no data loss). With this step, the probe stops capturing the flow, produces it to the classification step discussed ahead and creates a new flow with the same identification (i.e same IP addresses and ports). In other words, the goal of this step is to break flows into smaller fragments to avoid manipulating very large flows (e.g. flows that last hours to days). This prevents large flows from exhausting most of the ISP's storage space as older fragments can be discarded and aids in making them available for correlation sooner.

Flows resulting from the previous phase must now be classified into the respective Tor Flow type, or rather, CF or OSF, the first being related to the traffic between a client and the entry relay and the second to the communication between the exit relay and the OS. This is achieved by employing the separation models from the work of Lopes et al. [2], namely, the source and target separation models. It begins by extracting the necessary features that will serve as input for the models. The source separation model is the first to be applied, identifying the OSFs from the remaining flows, followed by the target separation model where the CFs directed to the clear web are distinguished from those directed to an OS and thus concluding the flow formation pipeline.

Finally, there are some limitations associated with the current ISP probe pipeline for the *SUMo* case. First, the filtering phase is incapable of separating flows associated with Tor directory. Tor periodically accesses directory relays to fetch metadata regarding the Tor network. We could not find a method to distinguish such flows as not only a relay can be simultaneously a normal and a directory relay but directory flows are almost identical to CFs. As a result, CFs and OSFs become polluted and inaccurate correlations might occur. Another concern is the possibility of a single OSF containing multiple sessions from different clients simultaneously. This makes comparison more difficult as data from two or more sessions become intertwined within the flow, lowering the accuracy of the correlation model.

[1] https://www.dan.me.uk/torlist/?full

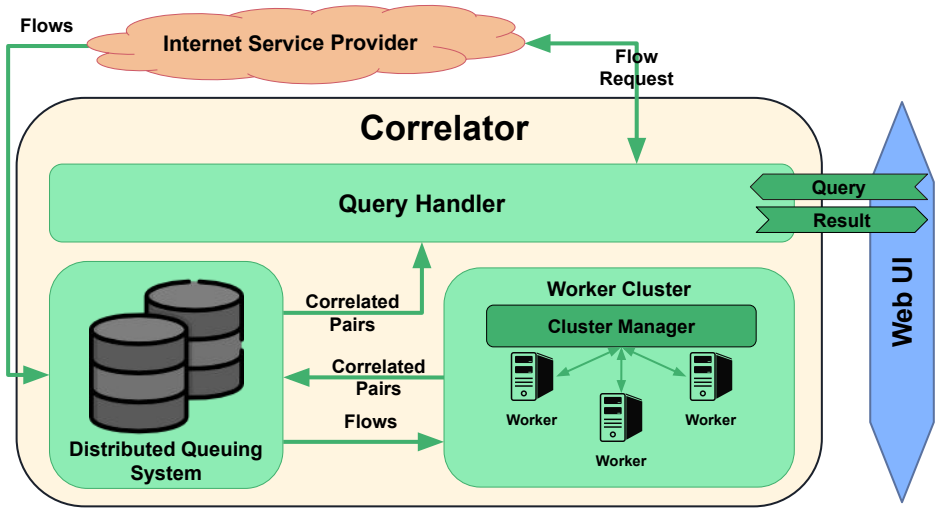


Figure 4.3: Correlator system architecture.

4.3 Correlator

Upon receiving a query, the Correlator often needs to handle high volumes of data obtained from the ISPs. This can arise from multiple factors, and among them are the potential presence of a significant amount of identified Tor Flows to process within at least one of the ISPs, the inclusion of numerous ISPs, or simply because the flow pair combination results in large amounts of data to be correlated. The original correlation model, as proposed by Lopes et al. [2], was introduced as a centralized solution, ensuring the potential for distributed computation across multiple servers. Therefore, this section will address how we can implement a distributed solution for the base Correlator described in Section 2.1.3, aiming to efficiently process these extensive amounts of information. Moreover, the Correlator will not be restricted to handling a single query per time slot, as our intention is to support not only simultaneous queries from a single user but also to accommodate multiple users accessing CorrForce concurrently. Another key feature is achieving a modular and flexible system. CorrForce's modularity stems from the independence of its components, facilitating easy upgrades. Additionally, CorrForce is flexible, allowing straightforward implementation of new correlation models for the use cases described in Section 2.2.

To achieve the desired distributed characteristics, we introduce the three modules forming the correlation component, respectively, *i*) the Query Handler (QH), *ii*) the Distributed Queuing System (DQS), and *iii*) the cluster of Worker Nodes tasked with the actual data correlation, all integrating the Correlator's architecture represented in Figure 4.3 along the interaction model among each other.

The first module, the QH, is the active element within the Correlator, responsible for the intake and breakdown of user correlation queries, as well as for requesting the necessary flow data from ISPs and managing the query execution process. This component possesses most of the control over the query execution process, as such, the remaining modules are passive modules that wait for instructions. In the current stage of development, this

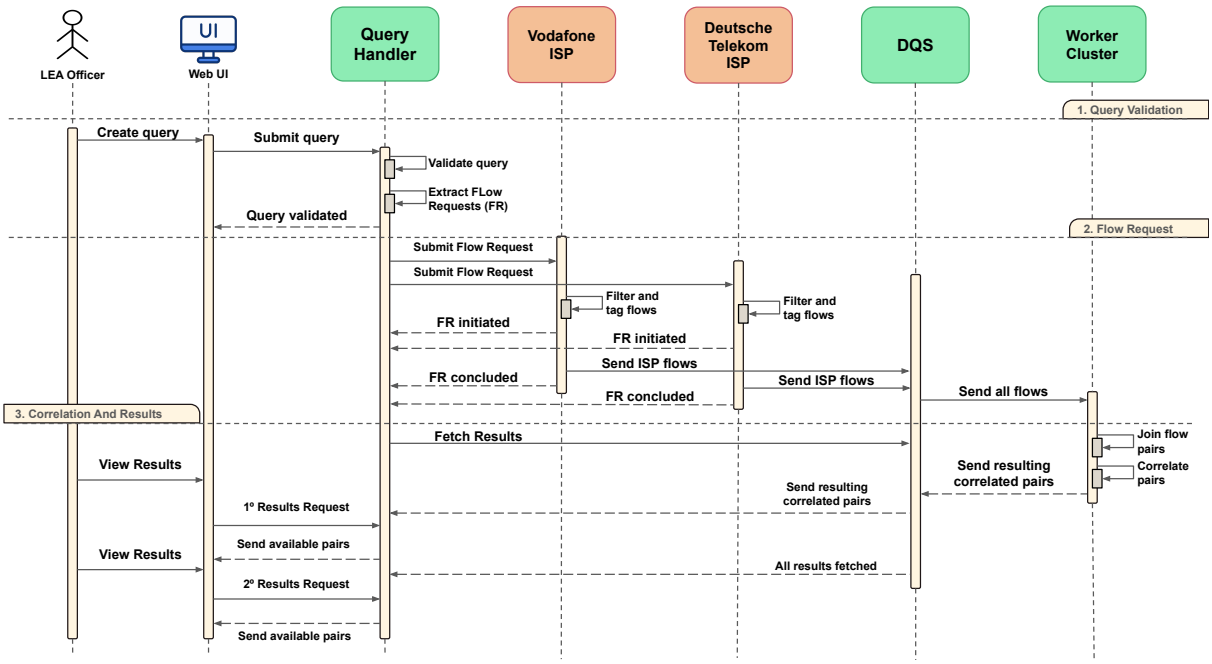


Figure 4.4: Time diagram of query execution.

module is a critical centralized server, i.e. running over the assumption of never failing, while having its work distributed across threads to support concurrent behavior when handling multiple queries.

Meanwhile, the DQS acts as an intermediate storage location distributed across a cluster, easing the communication and promoting isolation between the Worker Cluster, and both the ISP and the QH. The main feature for the DQS module is to play the role of queue. As numerous simultaneous queries can be issued, we face the challenge of managing their execution, therefore leading to the necessity of a queue. It will act in a bidirectional way, queuing flows from the ISPs and receiving the processed pairs from the Correlator. All input to the queue is processed in FR relative FIFO order, thus the flows of a FR are correlated in the ISP order, but if many FRs are issued simultaneously, the flows of different FRs will be processed in a mixed order achieving better work balance among FRs. Regarding this module's distributed features, we designed the cluster in order to achieve good availability, fault tolerance and eventual consistency [73], but we leave the details for this features to be discussed ahead in Chapter 5.

The Worker Cluster is the last module of the Correlator and tackles the challenges imposed by the distributed nature of the correlation. It is comprised of a Cluster Manager (CM) node and all the Worker nodes responsible for processing the flows stored in the DQS. The CM is the central component of the cluster, responsible for managing the available worker nodes, as well as the distribution of data across those nodes. It is worth noticing that the definition of queries given above is not observed by the CM, as flows from the various queries will be mixed. Rather, flows present in DQS queue are loaded in small batches and distributed across multiple tasks to be assigned to the Worker nodes. When available, Worker nodes will take on such tasks as instructed by the

```
{"query_id": 'hash(current_time) + user_id',
"isp_name": 'Vodafone',
"region_list": [],
"subnets": []
"start_timestamp": '2025:02:07:00:00:00',
"end_timestamp": '2025:02:10:23:59:59',
"correlation_tool": {"name": 'SUMo', "version": 1.0}}
```

Figure 4.5: Example FR derived from the query use case from Figure 4.8.

```
{"query_id": 'hash(issuing_time) + user_id',
"timestamp": '2025:02:08:12:00:00',
"origin_flow": {...}, // flow descriptor
"destination_flow": {...}, // flow descriptor
"score": 1.8384,
"correlated_tag": true}
```

Figure 4.6: Example of pair descriptor.

CM, employing multiple threads to process tasks in parallel, maximizing the utilization of the node's processing power. In other words, the respective set of flows is retrieved by each Worker node from the DQS, then proceeding on correlation pipeline by formulating the pairs, applying the correlation of each flow pair and lastly, inserting the processed flow pairs, along with the obtained score, back into the DQS. Considering the *SUMo* case, the pair correlation will consist of applying the bucketizing and sliding subset sum techniques presented by Lopes et al. [2]. Both the batching process, the task splitting and distribution mechanisms and correlation pipeline will be tackled in Section 5 when diving into the implementation details.

With all the Correlator's components introduced, we will explore the query execution pipeline. For a clear step-by-step description, it will be divided into three phases, 1) Query Validation And Parsing, 2) FR Processing, 3) Correlation And Result Gathering, all illustrated in Figure 4.4:

1. **Query Validation And Parsing:** Once a query is submitted to the QH, the first step requires analyzing the structure and validating the composing attribute values, based on the aspects presented in Section 4.4.2. While parsing the query attributes, all warnings and errors that may occur are immediately forwarded back to the user, independent of whether the execution proceeds or not (i.e. proceed with warnings or abort if errors occur).

   With the validated query at hand, we need to identify the participating ISPs, which can happen by identifying the ones listed under the ISPs attribute or, in the particular case where only the regions attribute is provided, by extracting the ISPs that act over the listed regions. All information regarding the ISPs is stored locally in the QH inside individual configuration files containing the ISP the probing policy described earlier in Section 4.2. Having the involved ISPs identified, we proceed with the creation of FRs, each having

the purpose of instructing a single ISP of which flows are required for the current query correlation stage. Under the circumstances of a single query, a FR is associated to exactly one ISP, however multiple FRs can be tied to the same ISP under certain conditions, such as when multiple correlation tools are provided or if a query is set to execute periodically over a time span, leading to new FRs per trigger. Figure 4.5 for instance, pictures one FR that would be created considering the first query usage from Figure 4.8. The attributes of a FR are comprised of a query id, used to associate the select flows to the respective query for easy combination with counterpart flows in the future steps, a region and subnet list, acting as a spatial filtration based on the capture region of the flows and the carried IP address, respectively, a start and end timestamp to define temporal boundaries for the flow selection, and the respective correlation tool, to identify what is the correct flow descriptor and respective flow features to selected to ensure correct execution of the correlation mechanism in question.

2. **FR Processing:** As soon as FRs are formed, they are forwarded to the respective ISP's FR API. The ISP handles the FRs by loading the matching flows from its local storage module according to the correlation tool flow descriptor, attaching each flow with the respective query ID as stated above, and then sending the processed flows to the Correlator's Distributed Queuing System (DQS) detailed ahead.

3. **Correlation And Result Gathering:** Finally, the QH is responsible for keeping track of all the active queries and respective FRs, waiting for their termination, as well as gathering and forwarding the resulting pairs to the respective query issuer upon request. Gathered correlated pairs follow the general pair descriptor exemplified in Figure 4.6, formulated by the ID of the query, the time of creation, detailed descriptors of both flows making up the pair, and the correlation score along with the tag that identify the pair as correlated or not. Meanwhile, users with submitted queries can request to visualize such pairs, even for unfinished queries, with presented results comprising only the currently gathered pairs.

4.4 User API

As users engage with CorrForce through the *Web UI*, queries become the central aspect of the system. In this section, we will delve into the structure of a query and outline the various attributes that compose it, leading to optimal usage of the correlation mechanism. Following that, we will explore the potential outcomes based on the values provided, and conclude by presenting several query examples to illustrate additional use cases for CorrForce.

4.4.1 Query Descriptor and Attributes

The queries instruct the Correlator to perform correlation based on specified attributes. Such attributes can be categorized into four types: identity, temporal, qualitative, and spatial, each explored in the topics ahead and illustrated in Figure 4.7 for better visualizing how they are arranged together.

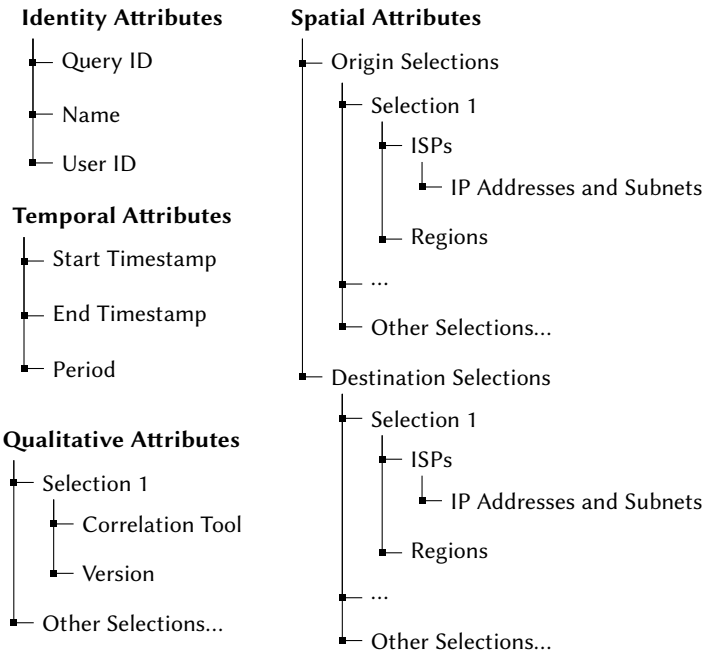


Figure 4.7: Query attributes.

Identity Attributes: With multiple queries flowing in from a range of users, distinguishing them becomes an essential task. The set of identity attributes serve this purpose by defining a unique identity for each query while linking it to the respective user. The uniqueness property is achieved through the combination of a randomly generated Query ID, for instance derived from the hash of the current timestamp, with the user's unique User ID. The additional Name attribute, allows the users to assign a brief, arbitrary name to the query for easier readability.

Temporal Attributes: Temporal attributes determine the timing of query execution, bounded by the Start and End timestamps, along with a periodicity, all using seconds as the time unit. These attributes serve two key purposes, namely, identifying when the query will run and specifying which flows to select, as outlined in the following cases.

With **both timestamps set in the past**, the query will execute exactly once as soon as possible, selecting only flows starting between the start and end date. Period attribute is ignored in this situation. If **only the Start timestamp is set in the past** (i.e. Start timestamp in the past and End timestamp in the future), additional executions will occur based on the Period attribute. The first execution will correlate all the past flows. Subsequent re-execution will update the Start timestamp to the current time, ensuring only the most recent flows are correlated. A similar procedure will take place if **both timestamps are set in the future**, however the query only starts on the provided Start timestamp and will re-execute according to the period, correlating the earliest flows until the End timestamp. In case **Start timestamp is left empty**, the query will assume the current timestamp and execute immediately, with possible re-executions depending on the remaining two attributes, while if **End timestamp is left empty**, a single query execution will be triggered at the Start timestamp, regardless of

```
{ "origin": [{
    "isp_list":[ {"name": 'Vodafone', "subnets": []}, {"name": 'Deutsche Telekom', "subnets": []} ],
    "region_list": [] }],
  "destination": [{
    "isp_list":[ {"name": 'Deutsche Telekom', "subnets": []}, {"name": 'Vodafone', "subnets": []} ],
    "region_list": [] }],
  "start_timestamp": 'None',
  "end_timestamp": 'None',
  "period": 'None',
  "correlation_tools": [ {"name": 'SUMo', "version": 1.0} ] }
```

Figure 4.8: Query usage case 1.

the Period attribute. Finally, with **empty Period**, the query will execute exactly once, either close to the End timestamp, if provided, to ensure all flows between the timestamps are correlated, or close to the Start timestamp if no End timestamp is specified.

Qualitative Attributes: These attributes determine the appropriate correlation mechanisms to be used. As we plan to support multiple correlation tools in future development, aiming not only at Tor but also at other systems (e.g. VPNs), we introduce the concept of a Correlation Tool. This attribute specifies the mechanism to be employed (e.g. *SUMo* [2], *DeepCorr* [9], etc.), paired with the version of the mechanism, if multiple versions are supported for that tool. A user can select multiple Correlation Tools simultaneously to act over the same attribute selection, allowing for different points of view from each correlation tool set of results.

Spacial Attributes: A query needs to specify both sides of the correlation (e.g. where to get the CFs and OSFs respectively on the *SUMo* scenario), therefore user selections are grouped into either Origin and Destination Selections. Within each side, the user can create multiple selections, each defining a range of ISPs and/or Regions (i.e. Continents, Countries, etc.). For instance, if only ISPs are selected all flows of the respective side type from those ISPs will be used in the correlation. When specifying a given ISP, the user can provide a subnet or range of IP addresses to filter the flows from the ISP. This in turn allows CorrForce's users to direct correlation efforts at some target. If only regions are supplied, then all the ISPs acting over those regions will be considered, however only the flows captured under the mentioned regions will be used. Finally, by providing both options simultaneously, the query interpretation mechanism will restrict the selection to the chosen ISPs, and respective IP range limitation, and act only on the respective flows if captured under one of the referred region.

4.4.2 Attribute Validation and Query Outcomes

While this query design allows for flexible customization, certain attribute combinations are not supported as they may lead to ill behavior. Therefore, it is essential to validate the provided attribute values when processing a query to ensure correct execution or, if necessary, a failure that does not disrupt the execution of the system. The validation process enforces restrictions on the following aspects, classified as either being **HARD** or **SOFT** constraints, depending on whether their violation leads to a complete or partial query failure.

```
{ "origin": [{
    "isp_list":[],
    "region_list": ["Portugal", "Spain"] } ],
  "destination": [{
    "isp_list":[ {"name": 'Verizon Fios', "subnets": [123.456.789.012/32]}  ],
    "region_list": [] } ],
  "start_timestamp": '2024:06:16:00:00:00',
  "end_timestamp": '2024:06:22:23:59:59',
  "period": 'None',
  "correlation_tools": [ {"name": 'SUMo', "version": 1.0} ] }
```

Figure 4.9: Query usage case 2.

Among the **SOFT** constraints, we identify, **ISP not supported**, since in order to access the flow data available from a particular ISP, CorrForce requires a set of authorizations for that particular ISP. If authorization documentation are non existent, this ISP will be excluded from the selection, allowing for the remaining query to proceed. In Section 4.2, when tackling the integration of ISPs within the query execution pipeline, the referred authorization procedure will be explored in detail. **Region or Sub-region not supported under ISP** occurs when specifying both a set of regions and ISPs, and it happens that some of the listed regions, or respective sub-regions (e.g. if selected Europe, Portugal is considered a sub-region), are not regulated by a particular ISP, leading to the omission of the respective combination. Likewise, **IP subnet not ruled by ISP** forces the exclusion of specified IP subnets that do not match to those associated with the paired ISP. The final **SOFT** constraint, **Correlation Tool or specified version not supported**, as the Correlator is equipped with a set of supported Correlation Tools, is applied if one or more of the selected tools does not respect the available ones, thus being removed from the list of selections. Additionally, a sub-case of this constraints occurs when selected tool, version pairs is supported by the correlator but not by one of the ISPs in the selections, with it being omitted from the request to such ISP. Each **SOFT** constraint applied during the query validation process leads to the formulation of a warning message with the purpose of informing the user of the query restrictions.

As for **HARD** constraints, **No selection under Origin or Destination Side** avoids executing any query that does not specify any selection on either the Origin or Destination side. Therefore, it is imperative that when creating a query both sides of the correlation are defined. Similarly, **No Correlation Tool**, sets the requirement for the selection of at least one valid Correlation Tool and version. Finally, **Start and End Timestamp not sequential** leads to query failure in the particular case of Start timestamp occurs in the future of the End Timestamp, due to the lack of logic. In a parallel to **SOFT** constraints, with each erroneous behavior on the query specification, an error is issued back to the user detailing the query abortion.

Successfully passing these restrictions indicates that the query is valid and ready for execution. As the correlation process progresses, the resulting flow pairs are incrementally forwarded to the *Web UI* where they are organized based on the correlation tool employed, with each flow in the pair clearly identified and characterized using a consistent descriptor format. More details will be given regarding the result formulation and structure when outlining the complete correlation pipeline in Section 4.3.

4.4.3 Query Usage Examples

CorrForce's query language is versatile, allowing its usage under multiple circumstances by combining attributes (introduced in Section 4.4.1) in different ways. To demonstrate its flexibility, we present three use cases. The queries are specified in *JSON* format and each tackle different requirements. All the use cases will be based on *SUMo* as it is the only supported correlation model at this stage of development.

Case 1: As a straightforward scenario, we revisit the example presented in Section 4.1, where a LEA agent issues a query to correlate Tor traffic between the *Vodafone* ISP and the *Deutsche Telekom* ISP. Figure 4.8 illustrates the selected attributes for this query. Two important points stand out. Firstly, the origin and destination correspond to the CFs and OSFs, respectively. Hence, to correlate the both flow types from both ISPs, each ISP must be provided in both origin and destination sections. Secondly, all temporal attributes are left empty (i.e. set to *'None'*), which leads to an immediate execution.

Case 2: In this scenario, a LEA either controls a honeypot OS or has information about the location of a real OS, revealing the its IP address (e.g. 123.456.789.012/32) and ISP (e.g. Verizon Fios). The objective is to deanonymize clients that accessed this OS from specific regions (e.g. Portugal and Spain) during a past period (e.g. from 16/06/2024 at 00:00:00 to 22/06/2024 at 23:59:59). Figure 4.9 depicts the chosen attribute values. Notably, the origin side only specifies regions, leading to all ISPs operating in those regions to participate in the query. On the destination side, the query narrows down to the specific IP address of the ISP. Lastly, since both timestamps are set in the past, the query executes immediately, analyzing only the flows within the specified time frame.

Case 3: This final case presents the opposite of Case 2, where a LEA controls multiple clients instead, operating within an entire public subnet (e.g. 123.456.789.0/24) managed by a specific ISP (e.g. Optimum Fiber). The goal is to determine if multiple accesses to a particular OS address matches the behavior of an OS running under specific combinations of ISP and region (e.g. Renater in Romania and Vodafone in Germany). The clients, and consequently the query, are scheduled to run between two dates in a near future (e.g. from 07/02/2025 at 00:00:00 to 10/02/2025 at 23:59:59). Figure 4.10 outlines the necessary attribute selections for this scenario, while it is worth noticing that, on the origin side, a subnet is specified to encompass a range of possible client IP addresses and, on the destination side, since two distinct combinations are required, separate selections must be created. Finally, the query is set to trigger every 24 hours, defined by the period value, for early visualization of results.

4.5 CorrForce Limitations

CorrForce's goal is to introduce significant advancements in live flow correlation. However, several limitations stem from its architecture and design choices. These limitations affect the system's effectiveness, availability, and security.

```
{ "origin": [{
    "isp_list":[ {"name": 'Optimum Fiber', "subnets": ['123.456.789.0/24']} ],
    "region_list": [] } ],
  "destination": [ {
    "isp_list":[ {"name": 'Renater', "subnets": []} ],
    "region_list": ["Romenia"]
  }, {
    "isp_list":[ {"name": 'Vodafone', "subnets": []} ],
    "region_list": ["Germany"]
  } ],
  "start_timestamp": '2025:02:07:00:00:00',
  "end_timestamp": '2025:02:10:23:59:59',
  "period": '86 400', // 24hours
  "correlation_tools": [ {"name": 'SUMo', "version": 1.0} ] }
```

Figure 4.10: Query usage case 3.

One major limitation is the dependency on ISPs for traffic capture. If ISPs refuse to collaborate, CorrForce lacks a critical source of traffic, rendering it useless. As a potential workaround, a new feature could be developed allowing LEAs to analyze traffic captured by their own means. This would require the current ISP probe component to be redesigned to fit LEA's traffic capture environment. Assuming that a fraction of the ISPs agree to collaborate, we still face the problem of having a partial network view. This could lead to incomplete and inaccurate results, as one flow of the matching pair may not be present in the analyzed set of flows.

Another concern is the potential abuse (intentional or not) by LEAs when submitting queries. Regarding the query language design introduced in Section 4.4, CorrForce only imposes restrictions on the range of parameters a LEA can select to ensure the correct query execution. However, there is lack on restrictions preventing abusive queries. For instance, a LEA can create a query that executes periodically every hour over a year or even one where all ISPs are selected, potentially causing a DDoS attack on CorrForce. Implementing stricter query controls and monitoring mechanisms could mitigate this risk.

Lastly, users of privacy preserving tools (e.g. Tor or VPNs) can still employ methods to avoid detection. One popular group of methods is Pluggable Transports (PTs) (e.g. obfs4, meek, Snowflake, etc). These tools allow users to obfuscate their traffic as defense against Internet censorship and surveillance. The techniques incorporated in such tools require an additional server named bridge that intermediates communication between the privacy preserving tool and its client, making the correlation process more difficult and complex. Using Tor's case as an example, the captured flow would only include traffic between a bridge and the entry/exit Tor node while identifying the *client-bridge* counterpart would require more sophisticated detection techniques capable of identifying and analyzing obfuscated traffic.

Summary

This chapter presented the design of CorrForce, including its architecture, the traffic probing system, the structure of the query language, and examples of user interaction. Specifically, the chapter detailed the three components required by CorrForce: the ISP Probe, the Correlator, and the user API.

For the ISP Probe, this chapter explained how a policy must be established with ISPs to describe the accessed traffic and the pipeline that the ISP needs to support. Regarding the Correlator, the chapter described the resource requirements and the necessary modules to be implemented. Finally, for the user API, the chapter explained the query system and the structure of CorrForce queries. The chapter ahead presents the implementation of all the aspects tackled here and the decisions made when selecting the technologies employed.

5

Implementation

Contents

5.1 Implementation Overview 40
5.2 CorrForce's Query Mechanism 42
5.3 Live Tor Flow Generation and Capture over a Simulated Network 45
5.4 Queuing Correlation Data through Apache Kafka 47
5.5 Distributed Correlation Over Spark Structured Streaming 49

Having the complete design choices defined we proceed to the description of how the desired CorrForce's architecture was reached and put to work. In this chapter we explore the technologies employed in each CorrForce component and the capabilities such tools possess, exposing the reasoning behind the implementation choices. The chapter begins with a description on the development environment in Section 5.1. Section 5.2 presents the implementation of the Query System in CorrForce. Section 5.3 follows with the explanation on how to simulate live Tor traffic and how to capture such traffic in the ISP probe. Then, Section 5.4 introduces Apache Kafka as the technology choice to integrate the DQS and explores all the configurations employed to achieve this module's requirements. Lastly, in Section 5.5, we discuss the implementation of the Worker Cluster responsible for the actual correlation process using Spark Structured Streaming and how the original *SUMo* algorithm was adapted to the distributed approach we follow.

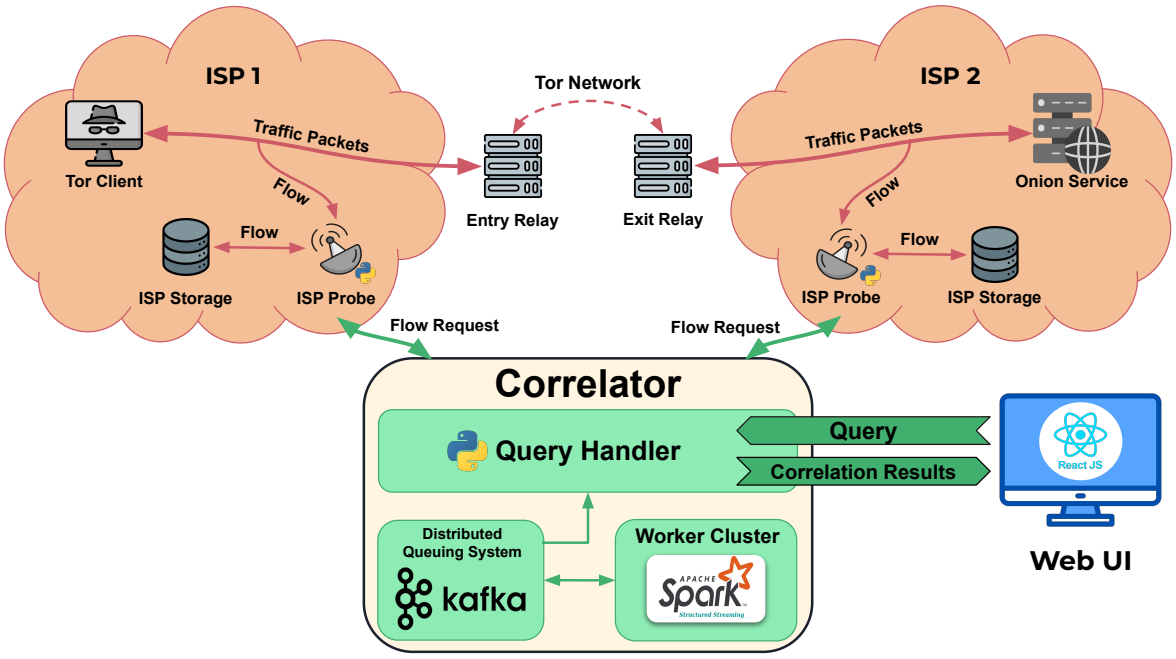


Figure 5.1: Implementation overview.

5.1 Implementation Overview

To achieve the desired features detailed in Chapter 4 for each component of CorrForce, we carefully and thoroughly considered the selection of appropriate technologies. This section explores the decisions behind these selections and discusses the characteristics of the chosen technologies that contribute to achieving the designed architecrure. For an overall view of the technology selection in CorrForce's implementation, Figure 5.1 presents all the system components along with their respective technologies.

Starting with the ISP Probing system, while including stages for traffic filtration, grouping, flow classification, handling FRs, and interacting with flow storage, it becomes essential to avoid introducing high levels of complexity into the pipeline. This ensures the required functionality is achieved while maintaining good performance. For this reason, Python was chosen as the implementation language, allowing us to seamlessly integrate these functionalities by leveraging its broad range of available libraries.

Addressing the query mechanism, which involves both the QH (from the Correlator module) and the *Web UI*, the primary desired characteristic is to provide a high degree of customization for user-issued queries. This is achieved through the previously defined query language, with the QH responsible for receiving, validating, and decomposing the queries accordingly. In addition to query formulation, the QH also handles features such as supplying data for user visualization of results, active queries, and ISP information. Similar to the rationale behind the ISP probe, we chose Python for implementing these functionalities due to its extensive library support, which helps avoid the complexity that would arise from integrating different technologies for each functionality.

However, as the number of functionalities grows, adding complexity to the QH becomes inevitable. Therefore, focusing on user experience is crucial to ensure that all features and information are easily accessible. To address this, the *Web UI* was created using React. React's front-end framework is widely used, providing numerous resources for the development of CorrForce's UI.

Let us now focus on the DQS. It fulfills the queuing mechanism requirements within the Correlator through Apache Kafka. Kafka is an open-source producer-consumer platform that integrates messaging, storage, and stream processing to facilitate the storage and analysis of real-time data. For the scope of this thesis, our focus relies solely on the first two capabilities. Designed to handle high volumes of data in real-time, Kafka efficiently routes data from producers to consumers, ensuring rapid and reliable delivery. It offers seamless integration and isolation between the two endpoint types. The distributed and scalable nature of Kafka endows it with high throughput, making it suitable for large-scale systems, such as LinkedIn's case [74]. Kafka demonstrates high throughput [75] while handling large amounts of real-time data and ensuring high availability, two properties we seek for the DQS module. Furthermore, Kafka is easy to integrate with Spark (i.e. the platform implementing the Worker Cluster, see Section 5.5). Numerous studies on real-time processing reported good results (i.e. high throughput, responsiveness and scalability) when combining both platforms [76, 77].

Regarding the last component, the Worker Cluster, we chose to implement it with Spark Structured Streaming, which is a powerful micro-batch stream processing engine. We took this decision after carefully considering the stream processing engines enumerated in Section 3.1.2. Spark emerged as the best option due to the following advantages: high throughput, making it well-suited for handling large volumes of streaming data efficiently; the widespread adoption in the industry, which translates into various sources of information and community support, boosting the integration of the desired features; support for a particular feature, stream-stream join, necessary for a successful implementation of CorrForce; excellent monitoring capabilities and easy statistics collection through its built-in *Web UI*.

Another reasonable option would be Kafka Streams, integrating seamlessly with the DQS as both would be based on Kafka platform. However, Spark stands out for its better support for complex stream processing tasks. Moreover, Spark also provides robust support for Kafka integration, ensuring smooth interoperability without any significant drawbacks.

For easy deployment and high portability, all components execute on Docker containers. Each component has a different custom Docker image equipped with all the necessary software requirements for its correct execution. Appendix A lists all images used.

Finally, although the system was designed to support multiple correlation technologies, the prototype developed for this thesis has implemented functionalities solely associated with *SUMo* in each of the system's components. Nonetheless, CorrForce was developed while having in mind a modular and flexible approach to facilitate the integration of future technologies.

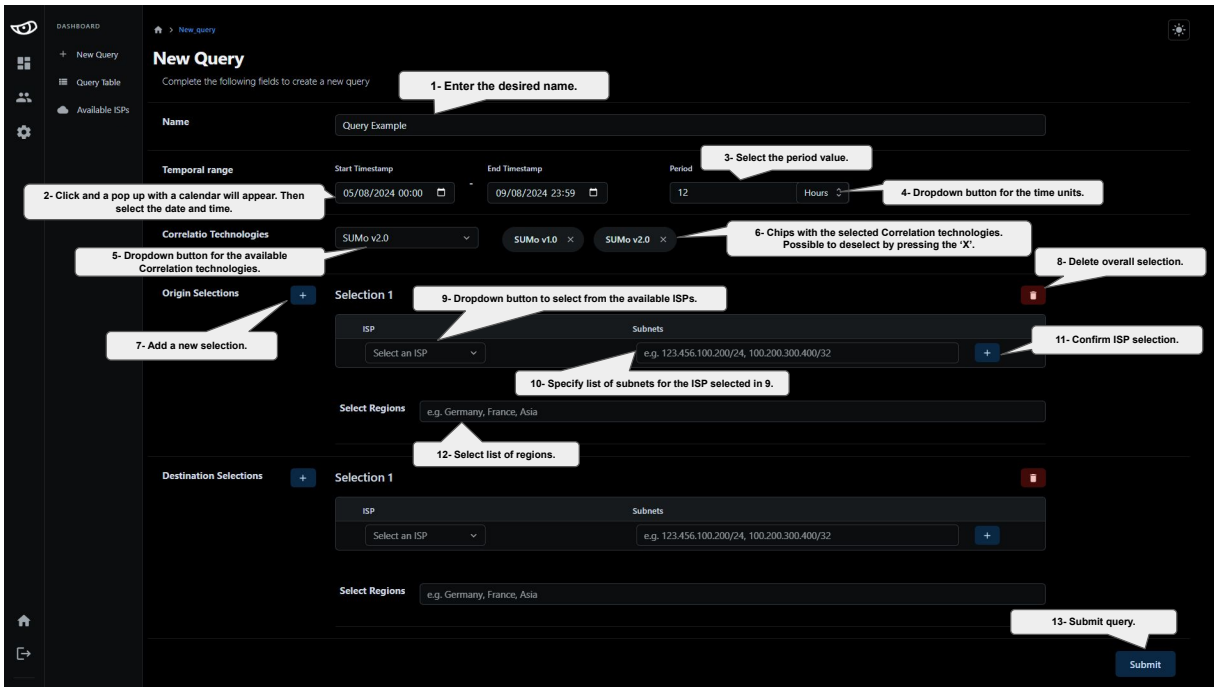


Figure 5.2: CorrForce *Web UI*: New query page.

5.2 CorrForce's Query Mechanism

The query system integrates within the QH (as stated in Section 5.1) carrying three main tasks: *i*) answer requests from the user; *ii*) manage queries and interact with ISPs; and, *iii*) receive and store the correlation results

CorrForce handles user requests through an API created with Python's *Flask* framework. The API supports submitting new queries, visualizing all submitted queries, viewing details of specific queries, and presenting information regarding ISPs or correlation technologies. To ascertain the correctness of each request, validation is applied using the *marshmallow* Python library[1], organizing and simplifying the required constraints and respective warnings and errors, by grouping related validations into classes. In particular, a new query is handled following the procedure constraints outlined in Section 4.4. Once validated, all FRs are extracted from the query. Given the scheduled and periodic nature of query execution, each of the derived FRs are likely required to be scheduled rather than executed immediately. Using Python's *schedule* library[2], the QH manages the timing of when FRs are forwarded to the respective ISP. Interactions with the ISP's probe FR API (see Section 4.2) are made using Python's *requests* library[3].

Finally, as an asynchronous periodic task, correlated pairs are fetched from the DQS using the *KafkaConsumer* from Python's *kafka* module, and immediately stored under the respective query in a *Postgres* database. This database stores correlation results, data regarding queries, ISP policies and available correlation technologies.

[1] https://marshmallow.readthedocs.io/en/stable/
[2] https://schedule.readthedocs.io/en/stable/examples.html
[3] https://pypi.org/project/requests/

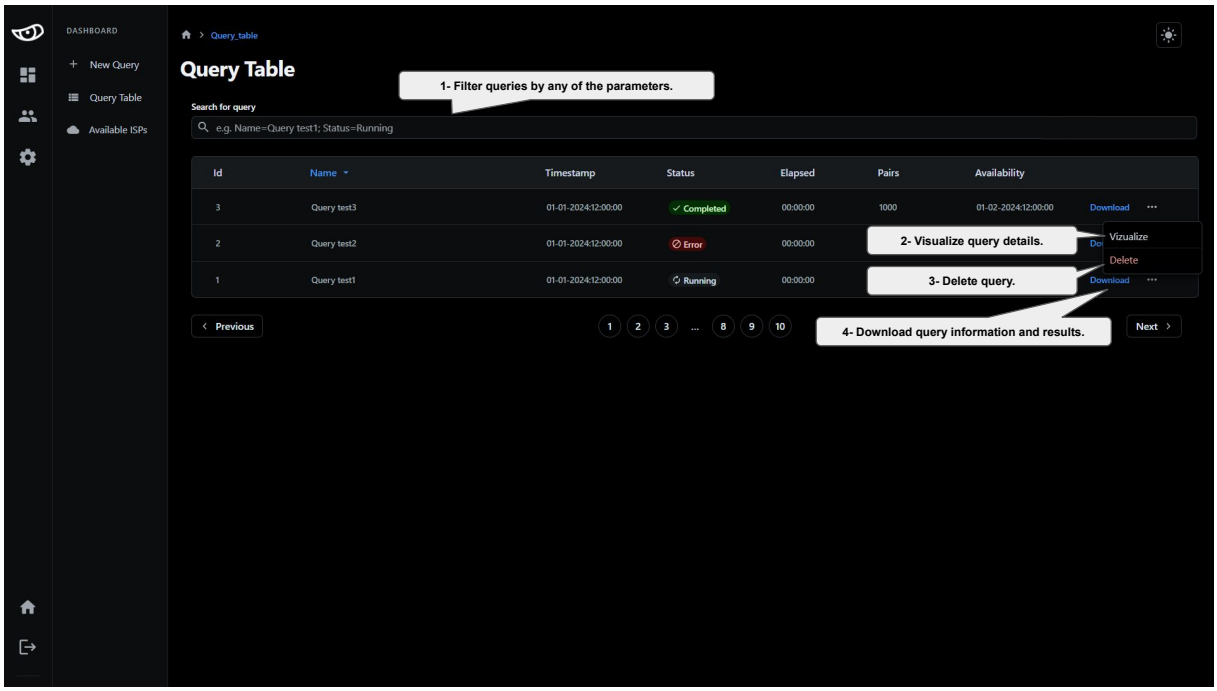


Figure 5.3: CorrForce *Web UI*: Query table page.

Users can interact with the QH through the *Web UI*. Figure 5.2 shows the *New Query* tab shown, which an user can access via the Dashboard. Among the required parameters to submit a query are: the name; the correlation tools; and, origin/destination selections. At least one Correlation Tool must be specified, selected by choosing one of the available options on the respective drop-down. Each correlation side must contain one or more selections, added by pressing the add button tagged with number seven in Figure 5.2. Within each new selection, the user must provide either an ISP, a region or both, in the their respective fields. If selecting an ISP, the CorrForce's operator can optionally list IP subnets in field 10 in Figure 5.2.

All submitted queries will be listed under the *Query Table* page from Figure 5.3, reachable from the Dashboard menu. Each row forming the table displays a brief summary of the query, containing the associated ID, name, submission time and date, status (i.e. Running, Error or Complete), the elapsed time, and if complete, the availability date, indicating the date and time until when the results will be kept in the database storage. For ease of navigation through the query table, the user can filter the entries based on the desired set of attributes, through the search bar placed above the table (e.g. Status=Complete&Timestamp<="01-01-2024:12:00:00"). Query details and obtained results can be visualized by pressing the *Visualize* button on the row's action menu. This action redirects the user to the *Query Details* page for that particular query. Within this page, the *Details* tab displays a summary of the query's submitted parameters, while the *Resulting Pairs* tab lists the results obtained as illustrated in Figure 5.4.

Looking at the latter's content, the first thing to be noticed is the correlated pairs table. For each output pair, a new entry in the table is added, detailing the pair ID, processing timestamp, correlation technology associated,

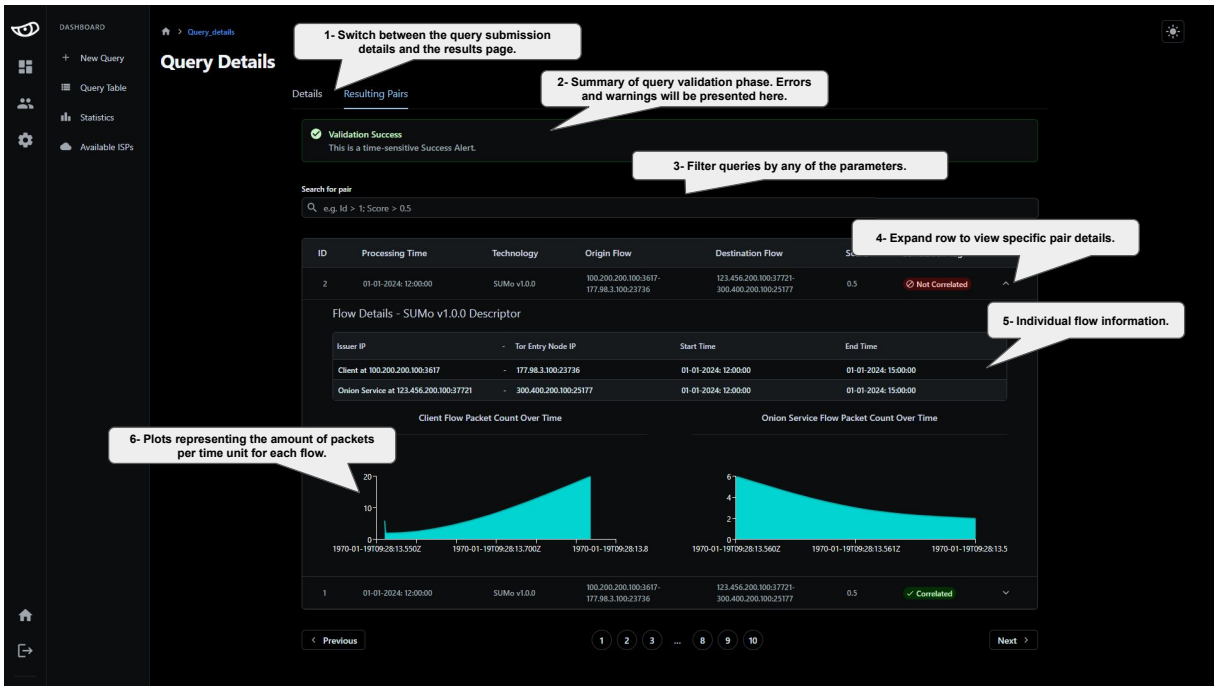


Figure 5.4: CorrForce *Web UI*: Query resulting pairs page.

composing flows and respective score and correlation tag. Expanding the row, more Correlation technology specific details from the pair are presented. We highlight the fact that this segment is expected to be different for each future Correlation model to be integrated in CorrForce. In the particular case of *SUMo*, the extra information displayed consists of each flow data as entailed by the *SUMo* flow descriptor, and a plot for each flow, representing the variation on the amount of packets exchanged between the flow participants per time unit. Moving to the top segment of the tab, the validation message displayed has the purpose of presenting the validation outcome, meaning that any warning or error that may occur will be shown here along with a small description.

Other non illustrated pages support additional features, such as exploring the available ISPs and respective policy data, and viewing the details of each Correlation Technology, are also made available to support the user when creating new queries.

Lastly, serving as future reference, some important features are left hollow in the *Web UI*. For instance, in the scenario a query is no longer useful to the user, it can be deleted from the system by pressing *Delete* in the action menu of the respective row in the table from the *Query Table* page. The expected effects of this action should result in complete erasure of the query from the QH, and if associated results are still arriving, they will be discarded. In the other hand, if the user desires to keep the obtained query results persistently, it will be possible to download all the associated information, including results, to the user's local storage.

5.3 Live Tor Flow Generation and Capture over a Simulated Network

As we incorporate *SUMo*'s correlation within CorrForce, it becomes imperative to have a reliable source of Tor traffic, to validate both the ISP probing mechanism and the correlation methodology. Therefore, in the scope of this section we entail the process of generating and capturing live Tor traffic within a simulated ISP network environment. With artificial clients, OSes, and ISP networks, we create a controlled environment to mimic real-world Tor usage, allowing us to live capture and analyze Tor flow, crucial for the testing and evaluation stages ahead.

5.3.1 Tor Traffic Generation

The first step towards generating Tor flows is to produce actual Tor traffic. At the current stage of CorrForce's development, we do not have access to Tor traffic from real ISPs. As such, we must create our own source of traffic to simulate a realistic environment. Using Docker containers, we can simulate clients and OSes and emulate their behavior when interacting through the real Tor network.

OS deployment: The OS setup begins by generating a unique onion address using the *mkp224o*[4] tool, along with a corresponding set of public and private keys, all crucial to the correct operation of the OS. In order to simulate realistic OS content, we utilize real OS web pages used in Lopes et al. [2] datasets. Upon deploying the OS container, the selected onion page is loaded along with the particular OS configurations (i.e. the onion address and public/private keys). On container startup, the Tor service is initialized, which deploys the OS within the Tor network, using the previously generated onion address and key set and configures the listening ports for the OS (i.e. port 8080 and 80), which will redirect all traffic to the web site serving location. Following this, the actual website is deployed through an *NGINX* server, which serves the selected onion page on port 8080 of the container, concluding the OS launching process.

Client deployment: Clients are simulated through Python, with each client script running within a Docker container. Most Tor clients typically use the Tor Browser, hence we simulate this behavior by creating a programmatic browser using the *TorBrowserDriver* from the *tbselenium* Python library[5]. The Tor Browser session starts by deploying the Tor service to ensure proper connectivity. The simulated client is provided with the list of all deployed custom OSes. During each browser session, the client can choose to access either these OSes or visit pages on the clear web. The clear web pages are selected from the Tranco list[6], which ranks the most frequently accessed websites on the web. Within each session, the client issues one or more requests to the selected OS or clear web page through the browser, interacting with random elements on the page, such as links and buttons. To enhance realism, we randomize the time interval between each request, to simulate users remaining on the current web page. In our traffic simulation, we only consider clients accessing one website at a time. The opposite

[4] https://github.com/cathugger/mkp224o
[5] https://pypi.org/project/tbselenium/
[6] https://tranco-list.eu/

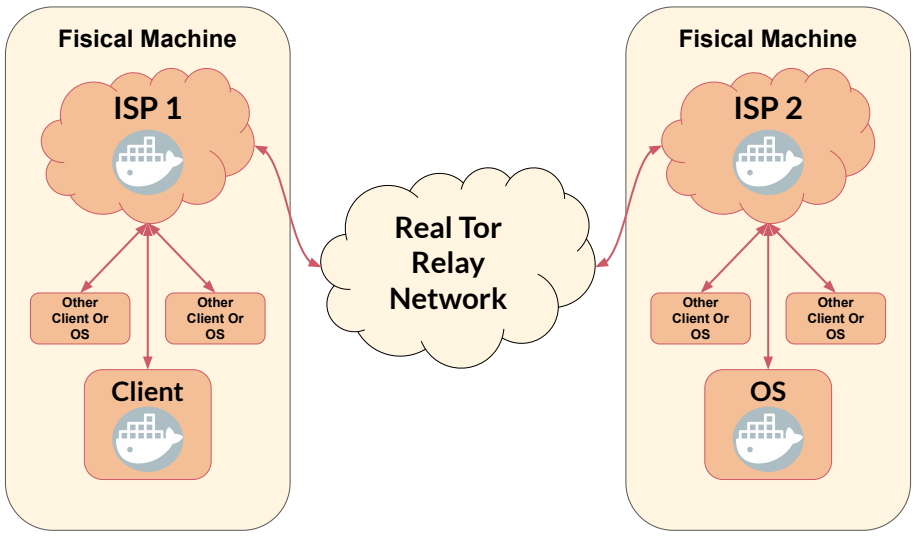


Figure 5.5: Simulating Client accessing OS with ISP flow capture enabled.

scenario would require extra steps to untangle the flows from each session, which to the best of our knowledge is currently unfeasible. Therefore, after completing a session, the Tor service is terminated to simulate clients closing the browser. This also forces the creation of new Tor circuits in subsequent sessions and prevents multiple sessions from using the same circuit, which is crucial for accurately distinguishing between individual client sessions for validation purposes. Finally, the client script can be configured through a set of parameters to attain specific characteristics of the client's simulated behavior (i.e. maximum stay time between requests, maximum session duration, probability of new requests vs session end, probability to access an OS or clear-net website, OS popularity, maximum number of Tranco pages).

5.3.2 Simulating ISP Networks with Flow Capture Capabilities

The following step describes how the ISP probe is implemented and how it can be deployed for online traffic capture as outlined in Section 4.2. The functionality of the ISP probe is achieved through Python, executing two concurrent tasks over threads: one dedicated to capturing Tor flows and the other responsible for classifying those flows, as described in the probe design and illustrated in Figure 4.2. Traffic capture in the first task is accomplished using the *Scapy* library[7]. Flow classification in the second is performed using the *SUMo* models[8] together with *Joblib* library[9] for model loading and execution. Classified flows are then placed in a simple *Postgres* database simulating the ISP storage. Such flows are exposed to the Correlator module through the FR API developed using Python's Flask framework. Upon receiving a request from the QH, the request is first validated to ensure it respects the agreement stipulated in the ISP policy. Soon after, flows are loaded from the ISP database

[7] https://scapy.readthedocs.io/en/latest/introduction.html
[8] https://zenodo.org/records/8366378
[9] https://pypi.org/project/joblib/

and uploaded to the DQS using the *KafkaProducer* from Python's *kafka* module, hence concluding the request procedure.

ISPs, clients, and OSes (deployed on individual containers as presented in Figure 5.5) interact through a shared Docker network. Client and OS traffic is then routed through the ISP container, using rules configured with the *Iptables* tool, and then forwarded to the real Tor relay network. As traffic reaches the ISP, it is captured and formulated into flows as described above. The simulation realism is enhanced by supporting the connection of multiple simultaneous clients and OSes to the same ISP instance. As a final side note, it is essential to ensure that the OSes and clients connecting to a specific ISP instance are launched on the same machine. This requirement raises from the need to share the same Docker local network among the containers for proper routing and traffic capture.

5.4 Queuing Correlation Data through Apache Kafka

The DQS module introduced in Section 4.3 is the distributed component responsible for intermediate the communication of the various ISPs and the QH with the Worker Cluster. As mentioned in Section 5.1, implementation is done over Apache Kafka's publisher/subscriber model, or simply Kafka's, as we will refer to from now on. Throughout this section, the focus will be directed to presenting the most relevant implementation details adopted when creating and configuring the Kafka cluster, where each component and configuration of Kafka is tailored to align with the prior design specifications for the DQS.

5.4.1 Cluster Implementation

Kafka's architecture is intricate, encompassing a multitude of concepts and components that work together to provide robust and scalable message queuing capabilities. First, we will explore the roles each play within a Kafka cluster, to then proceed into exploring the implementation choices we took, to create and configure them, based on the requirements for DQS module.

The Kafka cluster's central component is the broker. In other words, a cluster is formed by multiple broker services usually associate on a one-to-one basis with a server. Brokers, identified by an unique number ID, will not only share data among them but also act as a bridge between producers and consumers, and the Kafka service. Producers and Consumers will act according to their name and interact with Kafka Topics through messages. Topics are defined by their name and can be seen as a queue or stream of related messages. Handling such messages is done in a distributed fashion (i.e. through the brokers), as Topics are split in partitions, ranging from one to possibly thousands, scattered through the available Brokers. The higher the number of partitions the more parallelism can be reached by the producers and consumers (e.g. if a topic has 3 partitions, then 3 consumers can process messages to that topic in parallel). Each message is identified by an offset within their partition, and once a message is written it can neither be altered nor erased from disk (i.e. the message will persist until a given

retention time expires). In the current CorrForce implementation, we created a simple three broker cluster, with this number being the minimum as per the required fault-tolerance and consistency guarantees described next. However considering a future production scenario, more brokers should be used for better scalability.

Turning the attention to how data will be handled within the Kafka service. Defining the necessary Kafka topics according with the DQS design is the next step. Three topics are created at launch using a simple Python script with the *kafka-python* library[10]. Namely, the client-topic, the onion-topic and the correlation-results-topic, each with three partitions. The first two will handle CFs and OSFs, respectively, while the latter will receive the resulting pairs from the Worker Cluster. The messages carried in the topics (i.e. the flows and the resulting pairs) follow the Json format for easy serialization and de-serialization. Kafka clients (i.e. producers or consumers) can send or receive messages from these topics through brokers supplied in a list of bootstrap servers. All brokers in the cluster can act as bootstrap server as they possess the metadata of all the other brokers in cluster. The contacted broker will return necessary metadata gifting the client with the information of the broker containing the relevant topic-partition.

5.4.2 Achieving Fault-Tolerance And System Guarantees

As with any distributed cluster, DQS requires effective resources and metadata management, which was achieved though Apache Kafka Raft (KRaft), a consensus protocol based on Raft [78]. Introduced in recent versions of Kafka, KRaft surges as their own approach to tackle resource management, with the goal of avoiding the requirement for an external system, Zookeeper. Zookeeper was the default resource management prior to KRaft, possibly yet remaining the most commonly used due to being the default for years, marking presence in many systems [79,80]. However the KRaft decision was based on multiple drawbacks from Zookeeper. Namely, the requirement for an external distributed system deployed on separate nodes leading to separate installation, configuration and monitoring, hence increasing the complexity of the system, and the lack of support for high volumes of data. Additionally, Kafka developers have stated that the integration with Zookeeper is deprecated[11] in favor of KRaft.

Kafka deployment with KRaft is simple, requiring only the setup of a different type of node, the controllers, aside from brokers. We opted for integrating the controllers with both the role of controller as well as broker to avoid having more machines deployed, reducing complexity. Among the create controllers only one will be active at a time while the remaining will be waiting until set as active. However, a majority of these controllers must be alive in order for the system to be available. In other words, if 2f+1 controllers are defined, f+1 should be alive, hence the system is capable of tolerating f faulty controllers. In the CorrForce's case cited earlier with 3 controllers, we can tolerate one arbitrary controller fault. In case of extra brokers coming to be necessary, we maintain the current three configured controller/broker and add extra servers with only the broker role, since we believe that having three controllers is sufficient, assuming rare fault occurrences. Following the opposite line of thought, three controllers is the minimum we should have in order to tolerate at least one failing controller.

[10] https://kafka-python.readthedocs.io/en/master/
[11] https://kafka.apache.org/blog

Kafka relies on replication as the source of topic fault-tolerance, with the replication unit being the topic partitions. Partitions are defined in a set of replicas, with exactly one partition leader and possibly multiple followers. Leaders are always the most up-to-date replica as it handles all writes, while followers, acting as normal consumers, fetch the unreplicated messages to reach an identical version of the leader's log, in order to secure such data in case of a leader failure and to be capable of answering to client read requests.

The controller, as mentioned in the earlier topic, carries all broker metadata and manages the currently registered brokers, being capable of detecting when failures occur. The alive replicas (i.e. the non faulty) are defined by those capable of carrying regular updates to the controller (i.e. similar to an heartbeat messages) in addition to having a close log state to the leader's version. Such replicas are commonly referred to as "In-Sync" Replica (ISR)s. In the case of a fault occurring, the controller is responsible for electing a new partition leader to replace the faulty one. By having a dedicated machine for the election purpose (i.e. the controller leader), it is possible to group together elections for the various partitions affected, reducing the impact on availability. It is worth noticing that one broker may be responsible for multiple replicas of different partitions, hence, in a fault case, many partitions may be affected.

Moving forward, a dynamic set of ISRs is kept by Kafka and only such replicas can be eligible as leaders. In our approach, assuming three replicas, we configured a minimum of two ISRs to ensure that a message is only considered produced if the minimum ISRs acknowledge the message. This setup may impact the availability requirement of the system, however it ensures Eventual-Consistency by having some replicas persist the state containing the new message. Finally, with this ISR model, Kafka can, in the worst possible scenario (i.e. only the minimum ISRs have the message), ensure tolerance to f faults when setting the minimum number of ISRs to $f+1$.

5.5 Distributed Correlation Over Spark Structured Streaming

Being the core of CorrForce for the correlation process, the Worker is required to be robust and efficient when presented with high loads of data to correlate. In an attempt to meet this requirement, we implemented the cluster with Spark Structured Streaming. Throughout this section, we will explore the details of the Spark cluster configuration, followed by a description of the integration of *SUMo*'s correlation protocol within the cluster.

5.5.1 Spark Cluster Architecture and Work Distribution

Deploying a Spark cluster begins with the creation of the necessary nodes as illustrated in Figure 5.6, including a Client, a Master, and multiple Workers. The Master node is initialized by executing the *start-master.sh* script from the Spark tools package. Similarly, the Worker nodes are launched using *start-worker.sh*, also provided in the same package, but requiring additional parameters specifying the number of cores, memory allocation, and the address of the Master node. The Master is the node responsible for managing the cluster. Meanwhile, the Client node deploys the Spark application program. The code for this application is developed using Scala

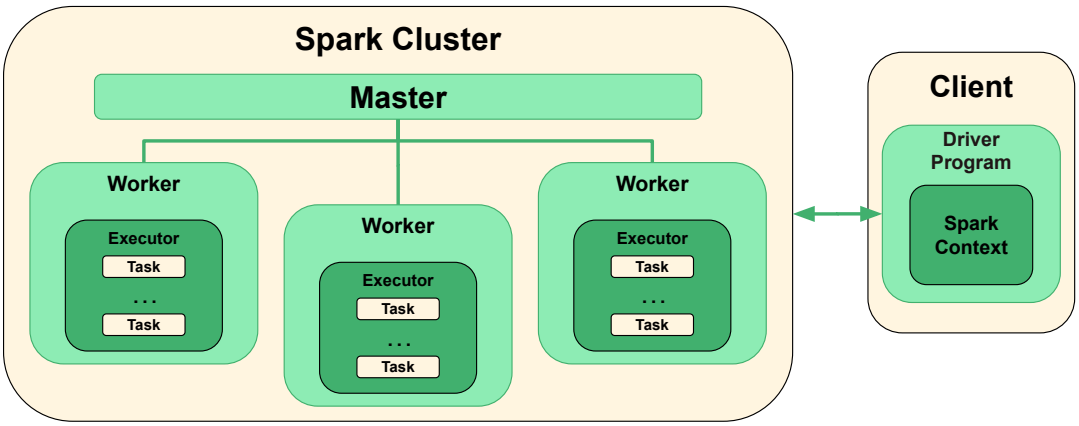


Figure 5.6: Spark cluster architecture.

language for performance reasons, with the remaining option, Python, likely resulting in slower execution times. The application consists of a *Driver Program* and *Executors* distributed across the Workers, and it is submitted by the Client through the spark-submit command (e.g. `spark-submit --class "com.spark.corrforce.CorrForceApp"--deploy-mode client --driver-memory 8g --master spark://($IP_{Master}$):7077 /path/to/corrforce.jar`)

The process running the main function of the application is named as the *Driver Program*, and is responsible for creating the *SparkContext*, which manages the available *Executors*. The *Executors* are the processes launched on worker nodes to run tasks and manage data across the node's memory or disk storage. Upon compiling the application code, managed through the *SBT*[12] project manager typically used in Scala projects, the output is a JAR file that contains the compiled Spark application along with its dependencies, excluding the Spark libraries, as these are already present on the cluster nodes.

Since we focus our development on Spark Structured Streaming, jobs are tightly coupled with the exact workload of each micro batch . Jobs are defined by multiple tasks, with the latter being the unit of work distributed through the executors for parallel processing. Tasks within a job are also grouped into stages, with some of these stages having dependencies on others. In our particular case, Spark automatically creates three stages. The first two related with flow retrieval from DQS (i.e. one for CFs and one for OSFs), and the latter with joining flows and correlating the pairs.

The deployment mode exemplified in the spark-submit command above declares where the driver process will be created. With *cluster* mode, the driver runs inside the cluster master, whereas in client mode, the driver is launched in a separate machine, such as the case of our application. For production environments, the deployment

[12] https://www.scala-sbt.org/sbt-native-packager/

Table 5.1: Spark application configuration.

Spark Configuration	Value
spark.sql.session.timeZone	UTC
spark.sql.streaming.noDataMicroBatches.enabled	true
spark.sql.shuffle.partitions	60

mode should be switched to cluster mode, as it avoids creating the extra Client node. However, for the current development and testing phases, we are operating in client mode, since it facilitates monitoring the execution of the system and the statistics collection used in the evaluation chapter to follow.

As a final short mention, upon launching an application, Spark deploys a *Web UI* presenting multiple statistics and information on all of its components (i.e. executors, jobs, stages, tasks, etc.).

5.5.2 Data Checkpointing and Worker Fault-Tolerance

As described earlier in Section 3.1.2, Spark's robust fault-tolerance mechanism ensures reliability and resilience in large data processing workflows. With RDDs, the key component for this mechanism, data is divided into partitions, each processed in a task from a given job. In the event of a worker failure, Spark recomputes the lost partitions using lineage information kept by RDDs, which tracks the transformations applied to the original data.

However, the lineage information can become extensive, leading to large overhead of re-computation. To safeguard already processed data and to prevent infinite re-computations in case of faults, Spark integrates a checkpointing mechanism. Checkpointing involves storing intermediate states of RDDs to a reliable storage system, ensuring data is secured. We implemented the checkpoint behavior with HDFS. To avoid adding even more complexity to the whole CorrForce architecture, we consider HDFS nodes to integrate the worker cluster. In other words, the HDFS data nodes, responsible for storing the file data, will be executed in parallel with Spark workers in the same machine (i.e. one data node and one worker per machine). Similarly, HDFS name node will run in parallel with the Spark master, and manage all the data nodes. By following this approach we also minimize the impact of accessing checkpoint data, as it is already close to Spark workers.

5.5.3 Adapting *SUMo* for Spark Structured Streaming Deployment

The Spark application starts by creating and configuring the *SparkContext*. The configurations applied to our application, listed in Table 5.1, include several key settings. The *spark.sql.session.timeZone* ensures consistent date formats across all workers, even if operating in different time zones. To prevent unnecessary processing, the *spark.sql.streaming.noDataMicroBatches.enabled* option is enabled, thereby avoiding the creation of empty micro-batches. Additionally, the *spark.sql.shuffle.partitions* parameter is set to an empirically determined value, which will be further explored when discussing the optimizations applied to the Spark pipeline.

Furthermore, a custom listener is created to record batch statistics, which will be utilized alongside data collected from the *Web UI* in the forthcoming evaluation section. As discussed in Section 5.4.1, the *kafka.bootstrap.servers*

Listing 5.1: "CorrForce Spark application main function."

```
1  function main:
2      client_flows ← read_kafka_topic("client_topic")
3      os_flows ← read_kafka_topic("onion_topic")
4      results ← kafka_write_stream("correlation-results-topic")
5
6      pairs ← join(client_flows, os_flows)
7      for p in pairs:
8          b_client, b_os ← generate_bucktized_packets(p_client, p_os, B_interval)
9
10         w_client ← get_sliding_windows(b_client, W_size, W_overlap)
11         w_os ← get_sliding_windows(b_os, W_size, W_overlap)
12
13         scores_window ← [] #Scores per window
14
15         for cw, osw in zip(windows_client, windows_os):
16             scores_window ← scores_window + [calculate_window_score(cw, osw, Δ, W_size)]
17
18         p_score ← calculate_final_score(scores_window, threshold)
19         results ← results + [p]
```

option is configured with the list of available Kafka brokers, and the *startingOffsets* parameter is set to "latest" to ensure that the application consumes only the most recent data.

Regarding the main functionality of the correlation mechanism described in Listing 5.1, two read streams are established: one for consuming data from Kafka's client-topic (Line 2) and another from the onion-topic (Line 3). Spark's functionality revolves around Datasets and their respective transformations. Hence, these streams can be viewed as continuously updated RDDs that undergo transformations to reach the final correlated pairs. From the final pairs, we create a write stream connecting back to Kafka through the correlation results topic. Notice that, although continuous processing mode is made available, as noted in Section 3.1.2, it currently does not support aggregation operations (e.g. join operation), therefore, we rely on the default micro-batching mechanism. In our particular case, batches are configured to trigger at a minimum interval of 5 seconds, however if one takes longer than that, subsequent batches will be delayed and triggered as soon as the previous concludes.

Following a detailed explanation of the process by which the two streams are transformed into a final stream of correlated pairs, we will look into how the flows are joined to formulate the pairs and then how pairs are correlated. Figure 5.7 complements the previous Correlator illustration, with the Spark application pipeline we have been describing, along with the integration of the remaining CorrForce components (i.e. DQS, QH, etc.).

Joining Flow Pairs: To generate the potential flow pairs, we perform a Stream-Stream inner join on the previously described streams (Line 6). The goal of this phase is to obtain all the pairs to be correlated. For Stream-Stream joins, Spark imposes the use of watermarking on each stream, over a time column. With watermarking, entries older then a predefined value are discarded. Spark also requires that such time column values take part in the join condition. These requirements are designed to bound the joinable results within a specified time interval, thus preventing the comparison of new records with outdated ones and increasing the matching performance.

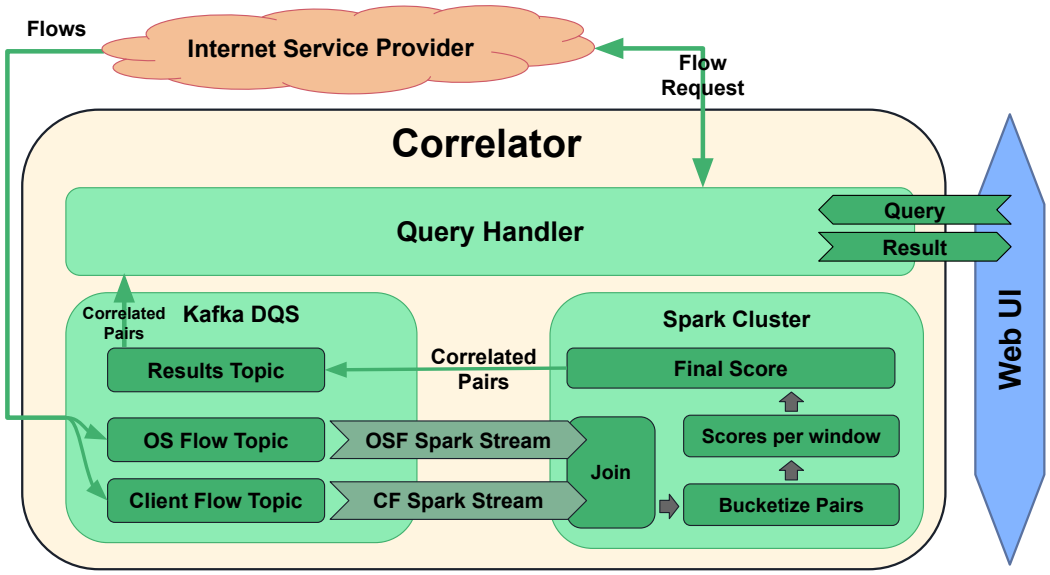


Figure 5.7: Complete CorrForce implementation.

Therefore, watermarking is applied to the initial packet timestamp of flows with a threshold of five days, meaning that any flow older than this watermark is discarded. In a first approach, the join condition would match flows that share the same ID_{Query} and have a difference in their initial packet timestamps of no more than seven seconds (e.g. for flows C and O, $C.query_id == O.query_id \& O.initial_timestamp - 7 <= C.initial_timestamp <= O.initial_timestamp + 7$). Once more, we enforce that the Worker Cluster is unaware of the query mechanism, hence ID_{Query}'s only purpose is to identify which flows should be matched together.

However, with this join matching condition, we would encounter workload distribution issues. Considering the scenario where a single query enters the system, all flows (i.e. Client and OS) will carry the same ID_{Query}. In Spark, data partitioning during a join operation is based on the key value used in the equality portion of the join condition, which in this case is ID_{Query}. As a result, all flows with the same ID_{Query} will be assigned to the same data partition, causing all the processing to be concentrated in a single task. In theory, this approach would enhance the placement of matching records on the same executors, hence avoiding data exchange between them. However, this approach undermines parallelism for a single query and, even in the presence of multiple queries, can lead to significant task skews due to the varying workload between queries.

To address this problem, we implement a salting technique as a workaround. The first step is to choose an appropriate salting range to maximize the cluster's parallelism (e.g. with three worker nodes each with three cores, the theoretically optimal salting range would be $3 \times 3 = 9$). The salting range determines the set of salt values that flows from both streams will take. For instance, with a salting range SR and flows with ID_{Query}, the salt values for these flows will range between $ID_{Query} \times SR$ and $ID_{Query} \times SR + (SR - 1)$.

Given that the OSF stream is expected to have fewer flows than the CF stream (i.e. CF can be directed to both

Listing 5.2: "SUMo correlation algorithm revised."

```
function calculate_window_score:
    sum_client, sum_os ← sum_values(client_window), sum_values(os_window)
    if ((sum_client, sum_os) == (0, 0)) return 0
    else if (sum_client > 0 || sum_os > 0) return -1
    else return sub_set_sum(client_window, os_window, sum_os, sum_client, Δ)

function calculate_final_score_version_code:
    seq_penalty, seq_gain, score ← 0, 0, 0
    num_windows ← size(scores_window)
    for s_window in scores_window[:-1]:
        if (s_window == 1) :
            seq_penalty ← 0
            seq_gain ← seq_gain + 1
        else if (s_window == -1) :
            seq_gain ← 0
            seq_penalty ← seq_penalty + 1
    if (scores_window[-1] == 1) score ← scores_window[-1] + (1 + 0.1 * seq_gain)
    else if (scores_window[-1] == -1) score ← scores_window[-1] - (1 + 0.1 * seq_penalty)
    score ← score / num_windows
    if (score >= threshold) return 1 else return 0

function calculate_final_score_version_alg:
    seq_penalty, seq_gain, score ← 0, 0, 0
    num_windows ← size(scores_window)
    for s_window in scores_window[:-1]:
        if (s_window == 1) :
            seq_penalty ← 0
            score ← score + (1 + 0.1 * seq_gain)
            seq_gain ← seq_gain + 1
        else if (s_window == -1) :
            seq_gain ← 0
            score ← score - (1 + 0.1 * seq_penalty)
            seq_penalty ← seq_penalty + 1

    score ← score / num_windows
    if (score >= threshold) return 1 else return 0
```

the clear-net and OSes), we duplicate the OSFs by the salting range and associate each copy with a different salt value from the determined range. For each CF, we attach a random salt value within the same range. We complete the modifications by using the salt value for the join's equality condition, instead of the ID_{Query}. This approach ensures that different portions of CFs will match with OSFs in separate tasks, thereby improving parallelism and balancing the workload across the cluster.

Finally, since Spark uses a hash partitioner for the join (i.e. data is placed into partitions based on the hash of the partitioning key modulo the number of partitions), we adjust the *spark.sql.shuffle.partitions* value from Table 5.1 to an empirical value higher than the salting range. This adjustment aims to minimize the likelihood of multiple salt values being assigned to the same partition. However, as we will discuss in the evaluation in Chapter 6, it is impossible to completely eliminate the aliasing of salt values without modifying the partitioner.

***SUMo* Correlation Algorithm Revised:** *SUMo*'s algorithm is the base for the pair correlation process in the current implementation of CorrForce. However, due to some inconsistencies in the original source code, which did not fully align with the algorithm as described, certain modifications were made to the source code to achieve a coherent version. Later on, regarding the Sub-Set-Sum problem inherent to *SUMo*, a different algorithmic approach is described that effectively improves the timing results of *SUMo*. Listing 5.2 presents the correlation algorithm functions called by the main function in Listing 5.1.

In the *SUMo* algorithm described in Section 2.1.3, CFs are paired using *incoming* packet times, while OSF pairs leverage *outgoing* packet times. These timestamps undergo a bucketization process (Line 8), where each bucket contains the packet count within fixed time intervals of $B_{interval}$. Sliding windows are then computed over the bucket list (Line 10) using window size W_{size} and overlap $W_{overlap}$.

An inconsistency arises here, as the original code discarded incomplete windows, not explicitly mentioned in the paper. This approach was adopted, to replicate Lopes et al. [2] results. The window score calculation proceeds as expected, applying the Sub-Set-Sum algorithm to the corresponding client and OS windows.

Another inconsistency, perhaps the most significant, arises during the calculation of the final correlation score over the window ones. For clarity, both versions of the final score calculation function are provided in Lines 7 and 22. The algorithm in the paper computes the final score using the sum of all window scores, with sequence penalties, $seq_{penalty}$, and gains, seq_{gain}, being gradually applied to each window score. However, the released code presents a final score derived solely from the last window score (Line 17), with the exact same penalties and gains being applied as in the previous version. We believe that the cause for this discrepancy was a bug where the variable for the score is "shadowed" by another variable with the same name in a nested loop.

As it will be discussed in Chapter 6, the two versions lead to different correlation outcomes, hence we proceed with the source code version to once more, achieve the same results gathered by Lopes et al. [2].

Finally, the original Sub-Set-Sum algorithm was neither optimized regarding time nor memory efficiency, hence, we implemented a new version tailored exactly to fulfill the specific needs of the *SUMo* use case. Listing 5.3 presents the pseudo-code for both versions of the algorithm.

In Line 1, the original approach makes use of a lookup table to validate the client window sum against the OS window sum. This table contains a boolean value for each number between 0 and the OS window sum, indicating whether it belongs to the set of solutions. The lookup table calculation in each execution instance requires $O(\text{sum} \times n)$ time complexity, with *sum* being OS window sum and n the window size. In contrast, the new approach (Line 18) adopts the exponential version of the algorithm, computing all possible sum combinations for the window's bucketized packet counts and leading to $O(2^n)$ complexity.

While the original version is generally considered more time-efficient, it is not suitable for the *SUMo* use case. Considering that in *SUMo*, windows are typically small and may contain large sparse values, it is likely that computing all values between zero and the sum becomes inefficient, when only a very small fraction of these values are solutions. For instance, considering a window of size 5 with a sum of bucketized packet counts reaching

Listing 5.3: "Sub-Set-Sum optimization."

```
function sub_set_sum_base:
    start, end ← sum_client - Δ, sum_client + Δ
    bufferSize ← end + 1
    lookup ← [false](2 * bufferSize)
    prev, curr ← 0, bufferSize
    lookup[curr], lookup[prev] ← true, true
    for i in [0, ..., W_size-1]:
      for j in [1, ..., bufferSize-1]:
        if (j < os_window[i]):
          lookup[curr + j] ← lookup[prev + j]
        else:
          lookup[curr + j] ← lookup[prev + j] || lookup[prev + j - os_window[i]]
      curr ← (curr + bufferSize) % (2 * bufferSize)
      prev ← (prev + bufferSize) % (2 * bufferSize)
    if ( any(lookup[start, end]))return 1
    else return -1

function sub_set_sum_optimized:
    start, end ← sum_client - Δ, sum_client + Δ
    SSsums ← []
    SSsums[0] ← 0
    current_index ← 1
    for i in [0, ..., window_size-1]:
      inner_index ← current_index
      for j in [0, ..., current_index-1]:
        sum ← SSsums[j] + os_window[i]

        if (start <= sum & sum <= end):
          return 1
        else:
          SSsums[inner_index] ← sum
          inner_index ← inner_index + 1
      current_index ← inner_index
    return -1
```

200, the original version would require $200 \times 5 = 1000$ iterations to calculate the lookup table. However, the optimized version would only need $2^5 = 32$ iterations to compute all sum combinations. This example highlights the major issues with the base version as, not only does it take significantly longer to compute results, but it also consumes more memory. The lookup table size can grow very large and is dependent on the input, while the sum combination list remains limited to 2^n entries regardless of the input size.

Summary

This chapter detailed the implementation of CorrForce prototype, including its different components and the chosen technologies. It details the infrastructure that ISPs require in order to integrate with CorrForce. More concretely, how the traffic is probed at the level of ISP's. The chapter also presents the interface through which CorrForce's users interact with in order to formulate and submit correlation queries. The next chapter evaluates

the results obtained with the system's prototype, regarding throughput, latency and accuracy of correlation results under multiple scenarios.

6

Evaluation

Contents

6.1 Evaluation Environment 60

6.2 Performance and Scalability 60

6.3 *SUMo* Reviewed Correlation Performance 67

6.4 Live Correlation 68

This chapter focuses on evaluating the CorrForce implementation, emphasizing system timing, scalability, and correlation accuracy. First, an overview of the evaluation environment is provided in Section 6.1. Section 6.2 presents the throughput results of conducted bench-marking on the various stages of the correlation pipeline, how the system scales with different Worker cluster configurations and how the overall system performs in terms of RT when correlating flows from a static dataset. In Section 6.3 we compare the revised version of the *SUMo* algorithm against the source code version. Lastly, in Section 6.4, we present the timing and correlation performance in a scenario with some realistic characteristics, where multiple clients and OSes are simultaneously deployed and live correlation is applied to the captured flows.

Table 6.1: Google Cloud resources per machine for static cluster evaluation stage.

Resource	Value
Machine Type	n1-standard-4
vCPUs	4
RAM Memory	15 GB
SSD Storage	80 GB

6.1 Evaluation Environment

In this section, we outline the evaluation environment. All nodes necessary to the execute CorrForce components were integrated within individual Docker containers, which allowed for a flexible and consistent environment across all the evaluation stages. The containerized approach not only facilitates testing, but also enables the deployment of CorrForce components on any machine that supports Docker.

We deployed CorrForce in Google Cloud using container optimized virtual machines. Table 6.1 presents the resources allocated for each virtual machine instance.

As we progressed with the evaluation, we noticed the complexity to launch the entire range of containers in each of the cloud machines increased. As such, we automated this process by means of a script using the Ansible playbooks[1]. This tool allows to launch all the containers in the correct order, as well as apply other environment configurations and run separate scripts in between container deployments. The script is written using the *YAML* structure which makes the contents easily understandable and modifiable if necessary.

6.2 Performance and Scalability

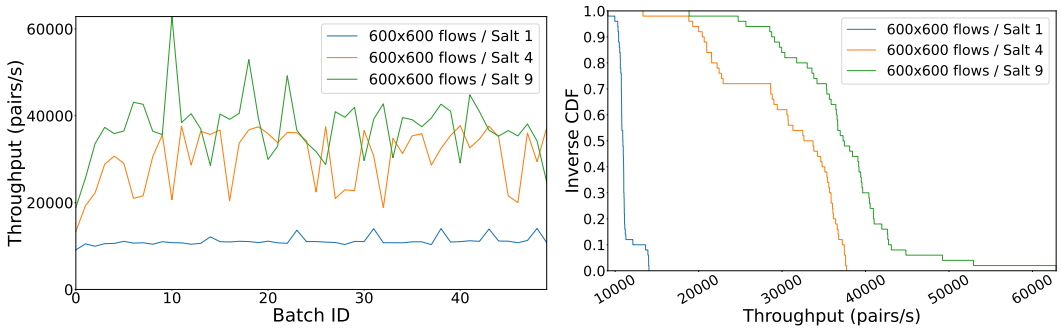
CorrForce's Spark Application pipeline is complex, with its performance influenced by various factors such as salt, input, and resources. In this section, we present micro-benchmarks that evaluate each stage of the pipeline and analyze the results to pinpoint the bottlenecks in CorrForce. Furthermore, we explore how modifications to the worker cluster configuration affect throughput.

6.2.1 Flow Join

Given the large volumes of flow data a query may involve, and that the number of pairs after the join grows quadratically with the number of flows, the join phase is a critical phase regarding performance. For this evaluation stage, we disable the correlation portion of Spark and focus solely on join performance. We consider two contrasting scenarios: one where all flows match under the imposed timing conditions, and one where no flows match. By examining these two specific cases, we can observe how Spark's inner mechanisms affect data distribution across workers and the pairing of flows.

We conducted synthetic executions to process exactly one query per execution across 50 executions. For

[1] https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html



(a) Throughput values as the number of pair/s across multiple batches.

(b) Inverse CDF of throughput values.

Figure 6.1: Flow join performance under $M \times M$ all-matching flows per stream.

test purposes, each Spark batch is considered as one execution. The maximum number of flows each batch can handle per stream is set to M. We extracted M flows of each type from the OS Test[2] dataset, capping each at 2000 packets. Then repeated this set of flows across batches without any variation, other then ID_{Query}. Finally, flows were inserted into their respective Kafka topics for processing.

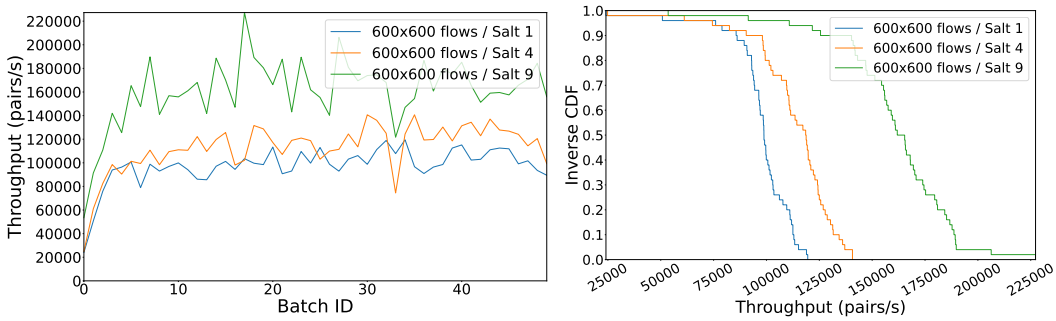
In this stage, we create a cluster configuration consisting of three worker nodes, each equipped with three cores. We also increase the default executor memory from 1GB to 4GB of memory for better results, however we will not explore the impact of this parameter. This setup was chosen to ease deployment while maintaining some degree of parallelism. For a given M, we evaluated the throughput of compared pair/s for each batch. It is important to note that the compared pairs differ from the output pairs of the join. For example, in a never-match scenario, despite the number of compared pairs being $M \times M$, the actual number of output pairs is zero. Additionally, we assessed the impact of the salting mechanism on workload distribution, as presented in Section 5.5.3, by varying the salting range used over a fixed number of 60 data partitions.

All match case: In this first scenario we force all CFs to match with all flows from OS within a batch. To achieve this effect, and considering the setup above, for each flow we altered the initial timestamp to always carry the same start date. This simulates an ISPs with many concurrent flows and ultimately leads to the expected worst-case scenario for our tool in terms of timing performance.

Figure 6.1(a) illustrates the throughput achieved over a span of 50 batches. Looking at the plot, the maximum average throughput observable is approximately 35,000 pair/s with a salting range of nine (i.e. the theoretical optimal range). Using the inverse CDF distribution, Figure 6.1(b) provides a clearer visualization of the throughput distribution for each salting range. For instance, with salting range nine, we can guarantee with 90% confidence that the throughput of the join phase will exceed the 28 500 pairs/s, when issuing a query to the system.

Secondly, we observe a significant increase in the average throughput between the executions with salting ranges of one and four, jumping from 10 000 to approximately 30 000 pairs/s, representing a threefold improve-

[2] https://zenodo.org/records/8369700



(a) Throughput values as the number of pair/s across multiple batches.

(b) Inverse CDF of throughput values.

Figure 6.2: Flow join performance under $M \times M$ never-matching flows per stream.

ment. However, given that the salting range quadruples, we could expect a proportional increase in throughput, which is not observed. This discrepancy can be attributed due to task skew emerging from different factors. The first factor is the duplication time of the OSF associated with the salting mechanism. However, its impact on performance is minor compared to the remaining factors, as we will observe in Section 6.2.4. The second, and most impactful factor, is the aliasing problem induced by Spark's default partitioning. With optimal partitioning, each salt value would correspond to exactly one partition. However, if Spark's partitioner maps two or more salt values from the OSFs to the same partition, then such partition will become larger. This impacts the batch processing time as it is determined by the longest partition. This issue is particularly evident when comparing the variances over the salting range one and four cases. In the first, there is minimal variation since all data is directed to the same task. In contrast, the latter case exhibits higher variation, reflecting the partitioning inefficiencies. The last factor is the randomness used to derive the salt value CFs. Scala's random number generator is not completely balanced and slightly contributes to partition skew increase. The final observation is that the impact of aliasing increases with the salting range. As the salting range approaches the number of partitions, the likelihood of multiple salt values being mapped to the same partition rises, favoring the aliasing effect.

Never match case: It follows a similar approach to the previous case, but instead of aligning all initial timestamps, we deliberately force all flows to be temporally apart. Figure 6.2(a) presents the throughput results through 50 batches for the never-match scenario, considering three different salting ranges. Comparing the results from this plot to Figure 6.1(a), the most noticeable difference is the higher throughput(i.e. on salting range nine, it becomes 4.2× higher, going from an average of 38 000 pairs/second to 159 000 pairs/second), indicating that non-matching pairs have less impact on the cluster workload. We put the hypothesis that the difference in the performance between the two scenarios is the result of underlying Spark optimizations discarding non-matching records earlier in the process. However, it becomes difficult to validate it as we could not find any information regarding Spark Structured Streaming join optimizations that conform with this hypothesis. Finally, with the throughput reaching the order of 100,000 pair/s, the variation in throughput increases proportionally, as

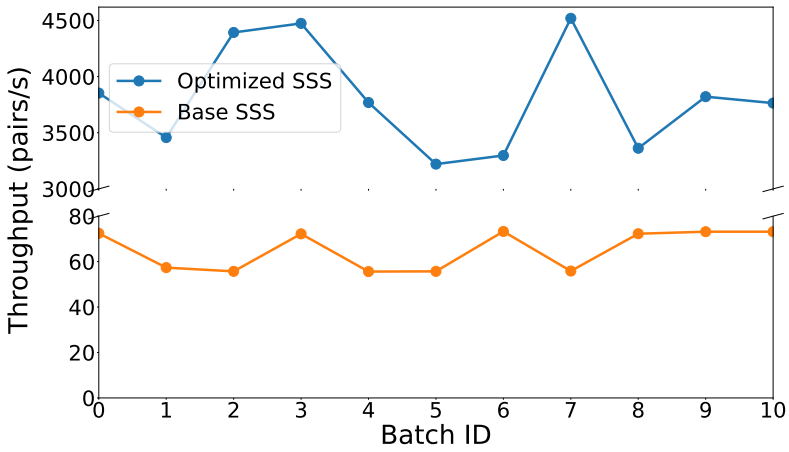


Figure 6.3: Full correlation pipeline performance comparison for 400 all matching flows of size 2000 packets between Sub-Set-Sum base and optimized version.

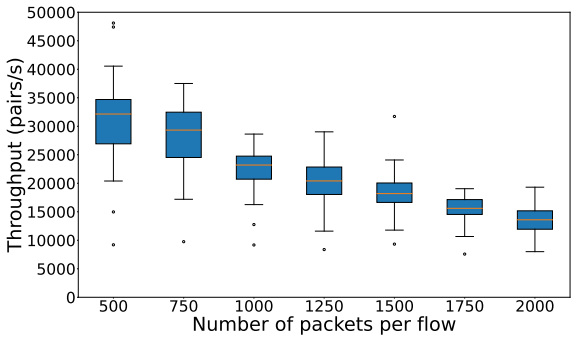


Figure 6.4: Throughput while varying the number of packets per flow under the 400 all-matching flows input.

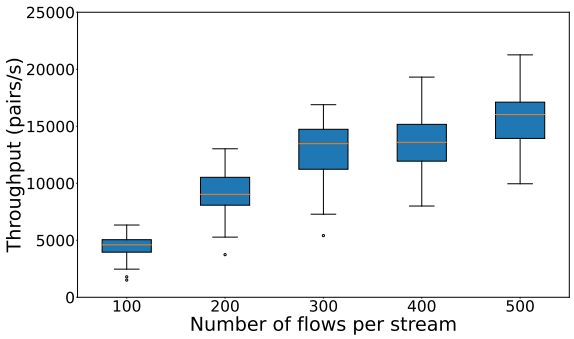


Figure 6.5: Throughput while varying the number of all-matching flows to correlate each with 2000 packets.

expected considering the observations from the previous scenario, reaching the order of 10,000 pairs/s.

6.2.2 Full Pipeline

Throughout this subsection we present the results of evaluating the complete correlation pipeline. Being a crucial stage, as it yields the final throughput values, we conduct multiple tests across a broad range of relevant scenarios to identify every nuance affecting performance. The cluster setup employed in this phase remains consistent with that of the Section 6.2.1 to facilitate comparability.

As a first step towards the full correlation pipeline assessment, we compare the throughput results of each variant of the Sub-Set-Sum algorithm, as described in Section 5.5.3. For this comparison, we used a salting range of one to allow for a direct evaluation without parallelism. Figure 6.3 illustrate the throughput of executing each version with 400 all-matching flows, each containing 2000 packets, derived through the approach described in Section 6.2.1. When comparing both versions, the difference in performance is evident, with throughput increasing from approximately 60 pairs/s to approximately 4000 pairs/s, indicating that the optimized version is

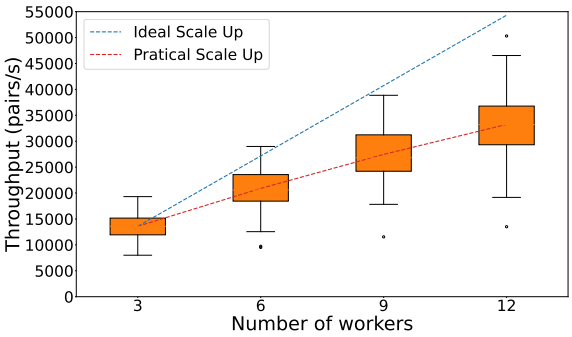


Figure 6.6: Throughput variation for different number of workers.

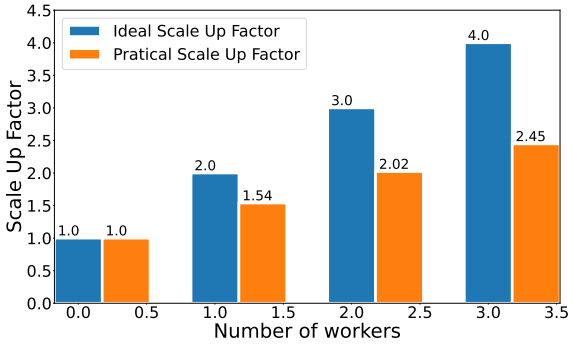


Figure 6.7: Ideal vs Practical Scale-Up factors from the baseline cluster of three worker node.

66× faster than the base version. This significant improvement is attributed to the substantial reduction in the number of sum combinations processed within the Sub-Set-Sum algorithm.

Focusing solely on the optimized version, we analyze the effect of two particular factors that impact the performance of CorrForce. Namely, the number of packets per flow, and the number of flows to compare. Figures 6.4 and 6.5 present the results of varying each factor, respectively. For each of the cases, box plots are employed to illustrate the variation of results over 100 executions, for each value on the x-axis. Additionally, the same salting range of nine is used for maximum parallelism in the current cluster configuration, in either case.

Regarding the first factor, we fixed the number of flows given to system in each execution to 400 all-matching flows. Then, within each execution, all the flows will share the same number of packets, however this number will range between 500 and 2000 packets between executions. As one can observe in Figure 6.4, the throughput reduces as expected with the increase in the number of packets per flow. The reason for this behavior is that, as the flow size increases, the number of bucket windows to be computed also increases, which results in longer executions of the Sub-Set-Sum algorithm for each pair.

Regarding the second factor, we induced the opposite situation fixing the the flow size to 2000 packets, and varying the number of all-matching flows between 100 and 600. From Figure 6.5, we would expect for the throughput values to remain close to constant, however we can see that the larger the number of all-matching flows the higher the throughput. What is observed is that with less data Spark does not put as much stress on the workers. In other words, Spark is faster with more data. We hypothesize that the reason behind these is to dampen the variation in throughput of the application in cases of large input data peaks. However to confirm this hypothesis, it would require a complex profiling of the worker nodes CPU usage, that we could not afford due to time constraints.

6.2.3 Cluster Scalability

To thoroughly evaluate the scalability and performance gains of the correlation pipeline, we executed the same set of tests across various cluster configurations. Only horizontal scalability will be evaluated due to time constraints,

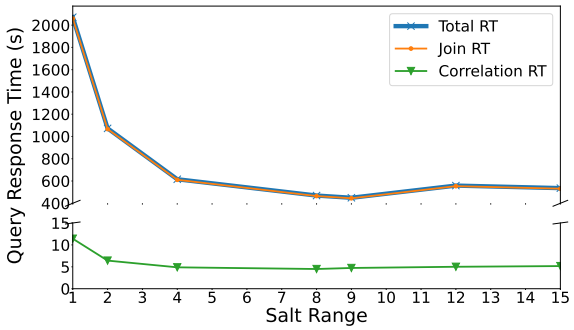


Figure 6.8: RT for a synthetic query over the OS Test dataset.

Number of CFs	14968
CF Max length	94407
CF Average length	3761
Number of OSFs	6888
OSF Max length	191785
OSF Average length	8502
Total compared pairs	103 099 584
Total output pairs (matched)	10 405

Table 6.2: OS Test query statistics.

hence the number of cores per worker will be fixed to three. Starting from the three worker nodes baseline, the number of worker nodes is incremented up to 12. This experiment aims to observe the performance gains, particularly the scale-up factor, as we increase the number of workers.

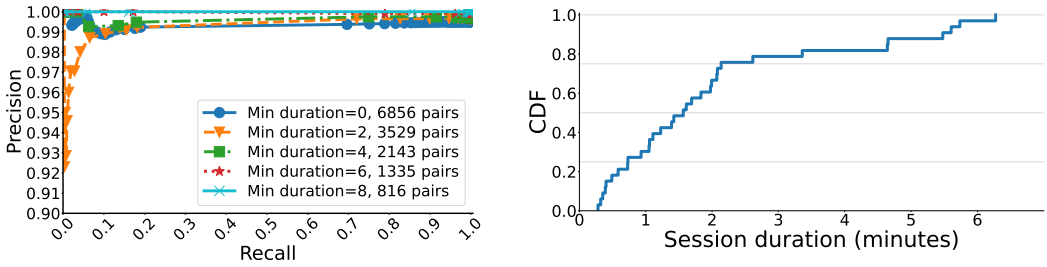
For each cluster setup, the following conditions are applied. The salting range is adjusted to ensure maximum parallelism (e.g. for nine workers with three cores each, the theoretical optimal salting range is 27). Results are based on the set of 400 synthetic all-matching flows of 2000 packets, used in previous phases, with the process repeated over 100 executions. Figure 6.6 presents the throughput distribution for the span of executions across each evaluated number of worker nodes.

As we increased the number of workers from 3 to 12, we observe a general trend of improved throughput, with performance gains relatively stable, demonstrating a near-linear increase. The practical scale-up factor, defined as the ratio between the throughput from the new and baseline cluster configuration, when compared with the ideal scale-up factor, formulated by the ratio between the number of cluster node, indicates the efficiency of parallelism in the cluster. Figure 6.7 illustrates the difference between the two scale up factors for each new cluster configuration. Finally, the scale-up gains are not optimal (i.e. 60-70% below the optimal value). Once again, this is mostly due to the aliasing problem we have identified earlier. Although we are allocating more computation resources, there will always be skewed partitions negatively affecting the performance. Their impact is larger when scaling the system and increasing the input.

6.2.4 Query performance Over Static Dataset

In the previous stages, our evaluation focused mostly on the throughput for the worst-case scenario, specifically when all flows matched. However, such an all-matching scenario is highly unlikely in real-world applications. To provide a comprehensive performance assessment, it is essential to consider a more optimistic scenario, where each CF only matches a smaller subset of the of the OSFs.

For this stage, we utilized the OS Test dataset to simulate a single complete query execution within the CorrForce. All flows in the dataset were tagged with the same query ID, to measure the RT taken to correlate



(a) Precision-recall curve for different threshold values considering a minimum session duration.

(b) CDF of false positives relative to session duration.

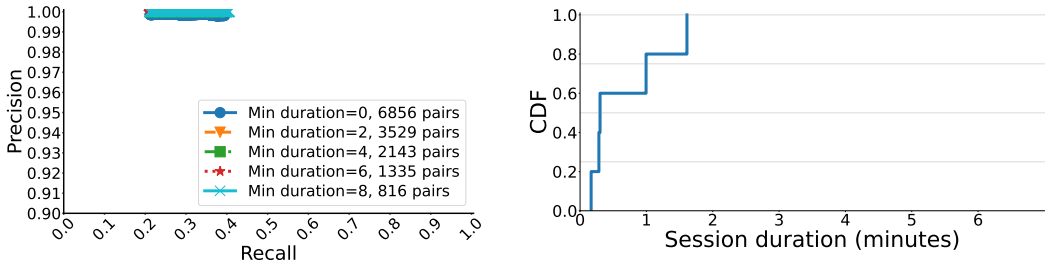
Figure 6.9: *SUMo*'s results with released code implementation.

all dataset flows. The query data is represented in Table 6.2, containing the number of CFs and OSFs input to CorrForce as well as the number of compared and output pairs. Looking at the values presented in the table, we notice that with this dataset CFs are matched approximately 0.7× with the OSFs (i.e. 14968 CFs compared to 10 405 output pairs), hence mimicking the smaller matching ratio we pretend.

Figure 6.8 displays the measured RT for flow correlation under different salting ranges, providing a clear illustration of how the salting mechanism effectively mitigates skewed tasks. The decreasing RT values for different salting ranges clearly demonstrate the effectiveness of the salting mechanism. We can observe a sharp decrease reaching the minimum at the salting range nine, with 5× less computation time then with a salting range equal to one.

Furthermore, when increasing the salting range past nine (i.e. the theoretical and observed optimal values), we can notice a 24.3% increase (i.e. from 449.43s with range 9 and 558.73s with range 12) in the RT relative to the salting range nine case. However compared to the range of values from smaller salting ranges, the increase is almost insignificant. From this observation we can conclude that the impact of the flow duplication incurred by the salting mechanisms is far smaller then the benefits to the parallelism of the system.

Finally, when decomposing the RT values between the join and correlation portion, as shown in Figure 6.8, we can observe a large discrepancy between the values, with the join taking almost all of the time (i.e. with range 9, the RT of join is approximately 94× the correlation RT). In scenarios as this one, where the number of matched flows is reduced, we estimate that the join performance will be the bottleneck. Although the impact of the correlation may not be as small, we can expect the join portion to be larger. A single CF will be compared to many OSFs available at the time, however it will only form pairs with those happening within a small time window of the CF.



(a) Precision-recall curve for different threshold values considering a minimum session duration.

(b) CDF of false positives relative to session duration.

Figure 6.10: *SUMo*'s results with paper algorithm implementation for the paper version.

6.3 *SUMo* Reviewed Correlation Performance

Given the modifications applied to *SUMo* regarding the inconsistencies found, the evaluation made in Lopes et al. is applied to assess the impact on the results of the correlation model. To ensure comparability between the two approaches, we used the original OS Test dataset developed by Lopes et al. [2], which contains 6856 correct sessions used in this evaluation stage. We also applied the same metrics used by Lopes et al. in the evaluation [2], namely: precision and recall. Precision measures the fraction of truly correlated flow pairs (true positives) over all pairs classified as correlated (true positives plus false positives). Recall measures the fraction of correctly identified correlated pairs (true positives) over all the actual correlated pairs (true positives plus false negatives). Additionally, the F1-score is used to derive the best precision-recall combination by balancing the two values through a harmonic mean.

Figures 6.9(a) and 6.10(a) present the precision-recall curves under the same range of thresholds (from -0.3 to 0.3) for the case with the released code version and with the paper's algorithm, respectively. Each figure also presents a curve when correlating Tor onion service sessions with different minimum duration.

Given the curves from Figure 6.9(a), for sessions with no minimum duration, *SUMo* can achieve a precision of 99.52% with a recall of 100% when maximizing the F1-score (i.e. threshold -0.3). This means that out of the 6856 identified pairs, all were classified as positive (i.e. 6823 true positives and 33 false positives). The figure also shows that *SUMo* can achieve higher precision values by limiting the minimum session duration, reaching 100% precision and 100% recall when setting the minimum session duration to six minutes.

Comparing the curves from Figures 6.9(a) and 6.10(a), the overall maximum precision slightly improved by approximately 0.3% (i.e. reaching 99.82% with threshold -0.3), but the recall experienced a significant decrease, going from 1 to 0.4 in the best case. Specifically, 2698 out of 6841 pairs are true positives (i.e. 4143 false negatives) and 10 out of the 15 are true negatives(i.e. 5 false positives). This suggests that with the current range of thresholds, while the number of false positives remains close to zero, the number of false negatives increased.

Figures 6.9(b) and 6.10(b) complement results with the CDF distribution of false positives according to the

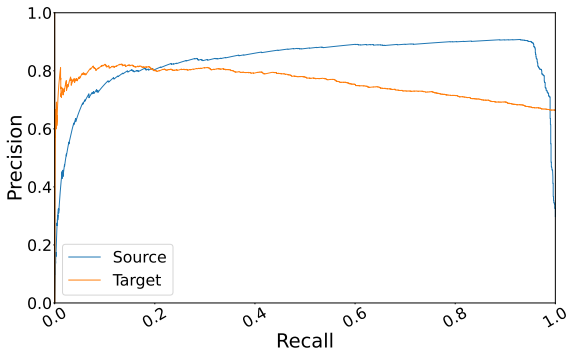


Figure 6.11: Precision-Recall curve for Source and Target separation models.

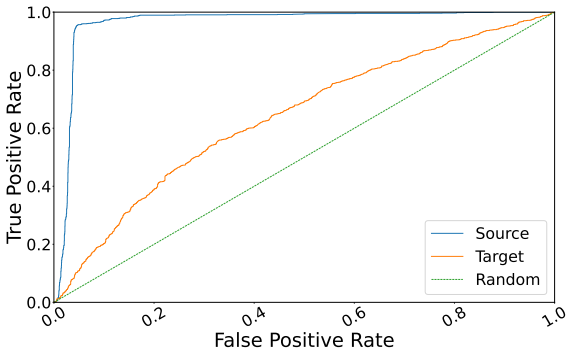


Figure 6.12: Receiver operating characteristic (ROC) curve for Source and Target separation models.

session duration with the optimal threshold (-0.3 in both cases) and both the released code and the paper algorithm version. The former indicates that false positives (i.e. 33 pairs) only occur in sessions with less than six minutes, with most occurring in sessions under 2.5 minutes, as noted by Lopes et al. [2]. In contrast, the latter figure shows that only sessions with less than two minutes are flagged as false positives (i.e. 5 false positive pairs).

Further conclusions regarding these results will be discussed in Section 6.4, as we compare them with those observed with the realistic flow capture from the ISP probes.

6.4 Live Correlation

The objective of this evaluation phase is to demonstrate the Correlator's ability to process flows as they are generated by ISP probes. The idea is to present the live interaction of CorrForce's components through the execution of a single query on ISP traffic traces set on the past. This query scenario is almost equivalent to one where the query evaluates ongoing flows, except for the overhead of live traffic generation (i.e. with only a few clients, it takes longer to generate the flows than to classify and correlate them). However, we ignore this problem, as in a real scenario, thousands of clients would be simultaneously active, generating flows more frequently. Given this objective, the experiment is divided into two parts: *i*) flow capture and classification, detailing the setup for generating and capturing realistic flows, and reevaluating the accuracy of the flow classification models from Lopes et al. [2]; *ii*) flow correlation, where under a query, CorrForce's ability to accurately match these flows is evaluated, alongside the assessment of query RT.

Flow Capture and Classification: For the first stage of live correlation, we employed a setup involving clients, OSes, and ISPs. Considering the same type of machines described in Table 6.1, each ISP machine as described in Section 5.3 is equipped with four clients/ OSes each in separate containers, and a container the probe. Given this configurations, we deployed six ISP machines. For easy validation of the captured flows, two of the ISPs will carry only OSes (i.e. for a total of eight OSes) and four of them will manage clients (i.e. for a total of 16 clients). Clients constantly generate requests to both the OSes and the clear-net, over a period of approximately two hours. To

simulate realistic clients, we applied the following configuration. We defined the maximum stay time between requests to two minutes, and the probability to establish a new session to 8%, hence we will have long sessions with relatively fast navigation. Clients have equal chance to access either an OS or a clear-net website, while the odds of selecting each OS are the same. Regarding the ISP probe, each captured flow is capped to 500 packets. After reaching 500 packets, the flow is stored. To avoid losing the remaining portion of the flow, a new one with the same session ID is created, and the trace capture continues normally. Considering the small window used in the join and that OSFs can be very large due to Tor circuit reuse between different client sessions, this approach minimizes the risk of missing pairs due to time differences. Collected flows are stored locally on the respective ISP machine, prepared for the correlation stage. By the end of the two hour generation period, we collected a total of 1623 OSFs and 3826 CFs, out of which 1285 are directed to the clearnet and 2541 to OS. Additionally, these are the result of 149 clearnet sessions and 378 OS sessions in total among all clients, for the ratios of 8.6 and 6.5 flows per clearnet and OS session, respectively.

By integrating the Source and Target separation models developed by D.Lopes et al [2] into the flow capture pipeline, we would be able to distinguish between OSF (positive class) and CF (negative class), and the CF directed to an OS (positive class) and the clear-net (negative class), respectively. During a first experimentation phase using the realistic flows, we noticed that the models were consistently classifying all flows as OS. This was the outcome of using the previously determined decision threshold values as they were biased towards the OS Test dataset traffic trends. For better interpretation of the the models classification performance, we reevaluate them over the set of realistic flows, and extracted optimal thresholds for the classification.

Figures 6.11 displays the precision-recall curve for each model, highlighting their ability to accurately identify the positive class. Figure 6.12 presents the ROC curves, which help us understand the models' capability to distinguish between the two classes. Tables 6.3 and 6.4 provide detailed statistical values, including the thresholds employed on the previous plots, the Area Under Curve (AUC)-ROC scores, and the precision, recall and F1-score, as well as the number of correctly identified flows of each type, using the optimal threshold.

Starting with the Source separation model, Figure 6.11 indicates consistently high precision (i.e. 80-90%), showing a low occurrence of false positives (i.e. few CFs classified as OSFs). Additionally, precision increases with recall, peaking at 0.9526 recall and 0.900 precision, indicating that the model can correctly identify OSFs while avoiding false positives. Figure 6.12 shows a near-optimal ROC curve (i.e. 93% less random then the random classifier), clearly showing that the model can achieve a high True Positive Rate (TPR) while maintaining a low False Positive Rate (FPR), indicating good separation between the two flow types.

On the other hand, the Target separation model results are less favorable. Figure 6.11 shows that precision decreases as recall increases, indicating a necessary trade-off between reducing false positives and identifying more positive class instances (i.e. client flows directed to an OS). Meanwhile, Figure 6.12 reveals a critical aspect as the ROC curve is relatively close to the random classification line (i.e. 28% less random), indicating that this model cannot accurately distinguish between the two classes.

Threshold Range	0.996 - 9.187e-05
AUC-ROC	0.9633
Best Threshold	0.1065
Precision	0.900
Recall	0.9526
F1-Score	0.92545
Correctly Identified CFs	3654 / 3826
Correctly Identified OSFs	1546 / 1623

Table 6.3: Source separation statistics over realistic flows.

Threshold Range	0.00612 - 4.68-e4
AUC-ROC	0.6412
Best Threshold	0.0004875
Precision	0.6643
Recall	1.0
F1	0.7983
Correctly Identified "To clear-net"	1 / 1285
Correctly Identified "To OS"	2541 / 2541

Table 6.4: Target separation statistics over realistic flows.

```
{"origin": [
    {
        "isp_list":[{"name": 'isp-3', "subnets": []}, {"name": 'isp-4', "subnets": []},
            {"name": 'isp-5', "subnets": []}, {"name": 'isp-6', "subnets": []}],
        "region_list": []
    }
],
"destination": [
    {
        "isp_list":[ {"name": 'isp-1', "subnets": []}, {"name": 'isp-2', "subnets":[]}],
        "region_list": []
    }
],
"start_timestamp": '2024:10:07:15:36:52', "end_timestamp": '2024:10:07:18:00:32',
"period": 'None',
"correlation_tools": [ {"name": 'SUMo', "version": 2.0} ]
}
```

Figure 6.13: Live correlation experiment query.

To derive the optimal thresholds presented in Tables 6.3 and 6.4, we employed the F1-score. The threshold yielding the highest F1-score was selected. For the Source separation case presented in Table 6.3, by employing the classification on the realistic set of flows, the model achieved 0.900 on precision and 0.9526 on recall, which positively reflects on the amount of correctly identified flows (i.e. 3654 out of 3826 CFs and 1546 out of 1623 OSFs). In contrast, the Target separation model, as expected from earlier observations, reached, under the optimal scenario, a precision of 0.6643 and a recall of 1.0. This is reflected on the correctly classified flows as all 2541 CFs directed to OS were correctly identified, but only a small portion (1 out of 1285) of CFs to the clear-net were correctly classified. This approach ensures that most of the relevant CFs to OS are identified, but it also results in a high number of false positives, as many CFs to the clear-net are misclassified. This trade-off, while ensuring that most CFs to OS reach the correlation phase, leads to an increased volume of "trash" flows, potentially impacting both query RT and correlation accuracy. Furthermore, an AUC-ROC of 0.9633 and 0.6412, respectively for the Source and Target separation models, summarizes the results of the ROC curves from Figure 6.12 and can serve as reference for comparison with future research.

Finally, considering the requirement to update the employed threshold values, we reaffirm the findings of Lopes et al. that frequent hyperparameter tuning is essential due to the changing traffic trends within the Tor network. However, implementing this in a live scenario poses challenges and can negatively affect model performance, highlighting the need for further research in live flow classification.

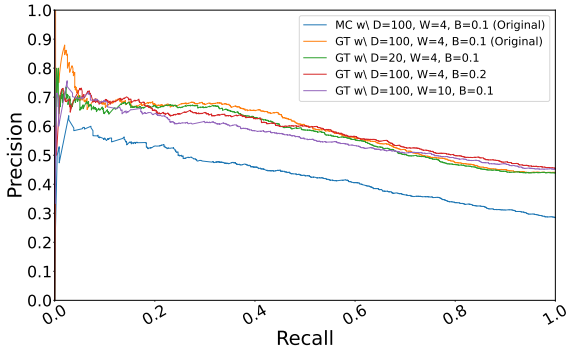


Figure 6.14: Precision-Recall curve for the correlation results with Ground Truth(GT) and Model Classified(MC) flows.

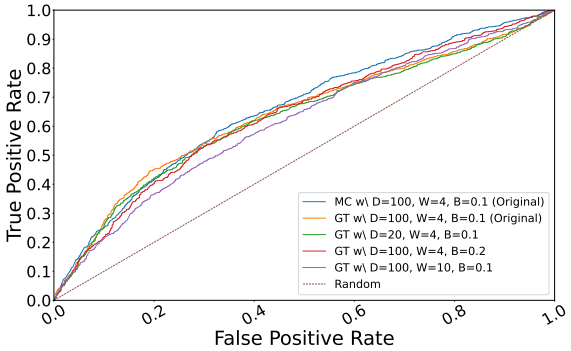


Figure 6.15: ROC curve for the correlation results with Ground Truth(GT) and Model Classified(MC) flows.

Flow Correlation: The next step towards live correlation involves defining the query required to correlate the flows captured during the previous stage. Figure 6.13 illustrates the query parameter definitions. The origin and destination selections include ISPs associated with the clients (i.e. ISPs three to six) and ISPs associated with OSes (i.e., ISPs one and two), respectively. For the temporal attributes, the start and end timestamps are set to the timestamps of the first and last captured flows, respectively, while the period is left empty as the data being correlated is set in the past. The correlation tool used is the reviewed and optimized *SUMo* version.

To assess CorrForce's capability to accurately correlate flows, we created a cluster with three worker nodes, each with three cores, and defined the salting range to nine for optimal RT. Given the observations regarding the separation models' performance, we need to evaluate the correlation performance using both the ground truth (i.e., flows in the correct positions) and model-classified flows. Figure 6.14 presents the precision-recall curve for the correlation results, highlighting CorrForce's accuracy when identifying the correct correlations. Figure 6.15 displays the ROC curves, illustrating the system's ability to accurately distinguish between correlated and non-correlated pairs. On top of this, each figure plots a line for both results over the ground truth (orange line) and model-classified (blue line) flows. Considering the ground truth case, we manually varied the correlation parameters to observe the effect on the accuracy of the correlation.

In an initial evaluation phase, we employed the same parameters for the correlation used by Lopes et al. [2] (i.e. window size as 4 buckets, window overlap as 2, bucket interval as 0.1s and delta as 100 packets). Through Figure 6.15, independently of the analyzed case, we make an early observation that the flow Correlator is not performing well in finding the correct flow combinations, as the ROC curves are very close to the random line (only 23.0% to 31.7% better than random). Figure 6.14, in either cases, shows that precision decreases as recall increases (i.e. reaching 44.14% precision and 99.82% recall on the ground truth case, and 40.47% precision and 60.87% recall on the model classified case, as per the maximum F1-score). This indicates a trade-off, as seen earlier with the Target separation model, between reducing false positives and identifying more truly correlated pairs. Additionally, by using the model classified flows, we can observe an expected lower precision (i.e. approximately 66%) when compared to the ground truth case, mainly due to the misclassification observed in the flow capture

Table 6.5: Average response time for the live query.

	Ground Truth	Model Classified
Total RT	20.972s	23.328s
Join RT	18.307s	20.343s
Correlation RT	2.665s	2.985s

and classification stage, introducing CFs to the clearnet that are not truly correlated with any OS flow.

Given that the results did not meet our expectations, we attempted to manually tune the correlation parameters. Each parameter was varied individually, setting the window size to 10 buckets, the bucket interval to 0.2 seconds, and the delta to 20 packets. For each variation, we performed correlation using the ground truth flows. Reexamining Figures 6.14 and 6.15, it is evident that these modifications did not produce significant changes in performance. Therefore, we make the conclusion that it is necessary to automate the tuning of the correlation model parameters in the future. This approach may yield better correlation results through the identification of an optimal combination of parameters.

Comparing these results with those obtained in the OS Test in Section 6.3, we identified the main factors that contributed to the deterioration of the observed results. In the static OS Test dataset, OSFs were captured in phase with their respective counterpart CF. This synchronization was achieved through artificial means where clients signaled OSes to start the flow capture. Since the join condition window is small (i.e. maximum of seven seconds apart between flows), only flows that start very close to each other will form pairs. Even with the levels of concurrency used to formulate the dataset, as stated by Lopes et al. [2], with such small join window, the formed pairs are mostly be the correct ones, leading to a minimal number of true negative pairs. This in turn results in a consistently high precision (i.e. surrounding 99%), as seen in Figure 6.9(a), given that the number of false positives is always low (i.e. 6823 true sessions vs 33 false sessions).

In the realistic flow capture scenario being presented, flows are no longer in phase as they are captured independently of one another. Furthermore, the flows captured in the live scenario are smaller (i.e. average 500 packets per flow, given the reasons presented in the beginning of the previous stage) when compared to the flows carried in the OS Test dataset (i.e. average 8502 and 3761 packets per flow for OSFs and CFs, respectively). Both of these factors not only influence the formulated pairs, but the discrepancy in flow length may also impact accuracy, as shorter flows provide less data for comparison.

Finally, we evaluated the average RT for this query under both scenarios with the ground truth and the model-classified flows. Table 6.5 displays the average total RT over 10 executions for each scenario, along with a breakdown of join and correlation RT. As anticipated, the model-classified case showed higher RT, specifically 11% greater (increasing from 20.972 seconds to 23.328 seconds). This increase is primarily attributed to misclassified CFs directed to the clearnet, which resulted in a higher number of pairs to evaluate. Moreover, the RT

breakdown reveals the significant discrepancy between join and correlation RT, previously noted in Section 6.2.4. However, due to the smaller number of flows generated in this scenario compared to the OS Test dataset (i.e. 5,449 vs. 21,856 total flows), the impact of the join RT is notably reduced (i.e. 87.3% of total time in the current case compared to 98.9% in the previous case).

Summary

The evaluation chapter provides a comprehensive analysis of CorrForce's performance and scalability. It begins by assessing the performance for each correlation stage of the pipeline and the scalability of the overall system. More precisely, we assessed the throughput of the flow join process, under different inputs. Complementing, the full pipeline is presented, highlighting the end-to-end throughput of CorrForce from flow join to correlation. Cluster scalability is tested, demonstrating close to linear scaling with additional resources. The query performance over the OS Test dataset is scrutinized, providing insights into the system's RT under optimistic conditions. The reviewed correlation performance of *SUMo* is evaluated, comparing the revised version against the original by measuring precision and recall. Finally, live correlation capabilities are tested, showcasing CorrForce's effectiveness in correlating traffic correlation scenarios. Although the obtained correlation results are not expected under realistic traffic conditions, the system is still capable of reaching low RT. The next chapter concludes this thesis by summarizing the key contributions and suggesting directions for future research.

7

Conclusion

Contents

7.1 Achievements 76

7.2 Future Work 76

Traffic correlation is a critical technique in network security and pattern analysis, with the core objective of linking disparate data flows to uncover hidden associations. This process involves analyzing various attributes of network traffic, including packet timing, volume, and routing paths, to detect anomalies and infer connections between different data streams. As we have seen, this method is crucial in areas such as intrusion detection, combating fraudulent financial activities, forensic investigations, and more.

In some of these use case scenarios, the correlation might involve processing large quantities of data within a useful timeframe. In this thesis, we have described CorrForce, a distributed solution capable of live correlating large amounts of data. CorrForce can utilize previously researched correlation tools, each targeted at distinct systems (e.g., *SUMo* from Lopes et al. [2] for Tor traffic correlation).

7.1 Achievements

The achievements reached in this thesis are the following:

1. We introduce CorrForce, the first query based live traffic correlation system, directed at performing large scale traffic correlation attacks, including novel ISP collaboration mechanisms for traffic capture, and a distributed module responsible for performing correlation tasks.

2. We presented a complete design for CorrForce, where we describe how a user can leverage our query language to easily perform correlation and how ISPs can aid our user's goals by contributing with their traffic data to CorrForce.

3. We listed and described all the technologies used to implement our design choices, providing detailed explanations and justifications for each one. Every component of CorrForce has been meticulously detailed, including comprehensive information on implementation and optimization.

4. We revised *SUMo*'s algorithm for correctness and introduced a new and more efficient algorithmic approach for the *SUMo* use case.

5. We evaluated CorrForce thoroughly, by individually assessing the timing of CorrForce correlation pipeline, scalability and the general capabilities to perform live correlation. Additionally, we performed a small evaluation on the correlation results of the revised *SUMo* version.

7.2 Future Work

The current version is tightly engineered, however there is a range of missing factors we will point out aiming at elevating CorrForce's potential.

The first and most important step towards such improvement is to achieve real-time query results, guaranteeing predictable query execution times. This new aspect of CorrForce will require the addition of multiple new features, such as developing methods to estimate query workload based on statistics from ISPs and data from previous queries. Implementing a dynamic worker cluster and resource manager capable of creating and destroying nodes to accommodate variations in input data will also be crucial. Additionally, employing techniques to prioritize relevant flows (e.g. most recent flows first, larger flows first, etc.) will enable earlier and more relevant results. Establishing a caching mechanism to access data stored in the QH database, relative to previous queries, will allow future queries to reuse previous results, avoiding re-computations.

Aside from achieving real-time capabilities, there is a need to fully implement the system with regard to security protocols, ensuring a secure channel between all involved parties. Furthermore, given that some correlation tools might be more computation heavy, it is important to equip worker nodes in the Spark cluster with

GPUs to handle these intensive tasks. Preparing the *Web UI* for production is also a necessary step to ensure a user-friendly and robust interface for end-users.

Finally, it is important to point out that we plan to integrate more traffic correlation technologies within CorrForce.

Bibliography

[1] R. A. Ariyaluran Habeeb, F. Nasaruddin, A. Gani, M. A. Amanullah, I. Abaker Targio Hashem, E. Ahmed, and M. Imran, "Clustering-based real-time anomaly detection—a breakthrough in big data technologies," in *Transactions on Emerging Telecommunications Technologies*, vol. 33, no. 8, 2022, p. e3647. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.3647

[2] D. Lopes, J.-D. Dong, P. Medeiros, D. Castro, D. Barradas, B. Portela, J. Vinagre, B. Ferreira, N. Christin, and N. Santos, "Flow correlation attacks on tor onion service sessions with sliding subset sum," in *Network and Distributed System Security (NDSS) Symposium*, 2024. [Online]. Available: https://dx.doi.org/10.14722/ndss.2024.24337

[3] A. Hartl, J. Fabini, and T. Zseby, "Separating flows in encrypted tunnel traffic," in *2022 21st IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2022, pp. 609–616. [Online]. Available: https://alexhartl.eu/papers/separating.pdf

[4] S. Miller, K. Curran, and T. Lunney, "Multilayer perceptron neural network for detection of encrypted vpn network traffic," in *2018 International Conference On Cyber Situational Awareness, Data Analytics And Assessment (Cyber SA)*, 2018, pp. 1–8. [Online]. Available: https://api.semanticscholar.org/CorpusID:54224969

[5] M. Stolikj, D. Jarnikov, and A. Wajs, "Artificial intelligence for detecting media piracy," in *SMPTE Motion Imaging Journal*, vol. 127, no. 6, 2018, pp. 22–27. [Online]. Available: https://www.researchgate.net/publication/326104447_Artificial_Intelligence_for_Detecting_Media_Piracy

[6] R. Dutta and K. C. Patel, "Detecting copyright violation via streamed extraction and signature analysis in a method, system and program," in *Google Patents*, 2008, uS Patent 7,363,277. [Online]. Available: https://www.freepatentsonline.com/7363277.html

[7] R. A. Ariyaluran Habeeb, F. Nasaruddin, A. Gani, I. A. Targio Hashem, E. Ahmed, and M. Imran, "Real-time big data processing for anomaly detection: A survey," in *International Journal of Information Management*, vol. 45, 2019, pp. 289–307. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0268401218301658

[8] R. U. Khan, R. Kumar, M. Alazab, and X. Zhang, "A hybrid technique to detect botnets, based on p2p traffic similarity," in *2019 Cybersecurity and Cyberforensics Conference (CCC)*, 2019, pp. 136–142. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8854561

[9] M. Nasr, A. Bahramali, and A. Houmansadr, "Deepcorr: Strong flow correlation attacks on tor using deep learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2018, p. 1962–1976. [Online]. Available: https://doi.org/10.1145/3243734.3243824

[10] I. Karunanayake, N. Ahmed, R. Malaney, R. Islam, and S. K. Jha, "De-anonymisation attacks on tor: A survey," in *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, 2021, pp. 2324–2350. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9471821

[11] S. He, Y. He, and M. Li, "Classification of illegal activities on the dark web," in *Proceedings of the 2nd International Conference on Information Science and Systems*, ser. ICISS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 73–78. [Online]. Available: https://doi.org/10.1145/3322645.3322691

[12] B. R. Hiraman, C. Viresh M., and K. Abhijeet C., "A study of apache kafka in big data stream processing," in *2018 International Conference on Information , Communication, Engineering and Technology (ICICET)*, 2018, pp. 1–3. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8533771

[13] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative api for real-time applications in apache spark," in *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, 2018, p. 601–613. [Online]. Available: https://doi.org/10.1145/3183713.3190664

[14] A. Iacovazzi and Y. Elovici, "Network flow watermarking: A survey," in *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, 2017, pp. 512–530. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7570208

[15] B. N. Levine, M. K. Reiter, C. Wang, and M. Wright, "Timing attacks in low-latency mix systems," in *Financial Cryptography*, 2004, pp. 251–265. [Online]. Available: https://users.ece.cmu.edu/~reiter/papers/2004/FC.pdf

[16] Y. Zhu, X. Fu, B. Graham, R. Bettati, and W. Zhao, "On flow correlation attacks and countermeasures in mix networks," in *Privacy Enhancing Technologies*, 2005, pp. 207–225. [Online]. Available: https://www.freehaven.net/anonbib/cache/flow-correlation04.pdf

[17] V. Shmatikov and M.-H. Wang, "Timing analysis in low-latency mix networks: Attacks and defenses," in *Computer Security – ESORICS 2006*, 2006, pp. 18–33. [Online]. Available: https://www.freehaven.net/anonbib/cache/ShWa-Timing06.pdf

[18] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning," in *MIT Press*, 2016. [Online]. Available: http://www.deeplearningbook.org

[19] S. E. Oh, T. Yang, N. Mathews, J. K. Holland, M. S. Rahman, N. Hopper, and M. Wright, "Deepcoffea: Improved flow correlation attacks on tor via metric learning and amplification," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1915–1932. [Online]. Available: https://www-users.cse.umn.edu/~hoppernj/deepcoffea.pdf

[20] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The Second-Generation onion router," in *13th USENIX Security Symposium (USENIX Security 04)*, 2004. [Online]. Available: https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router

[21] L. Basyoni, N. Fetais, A. Erbad, A. Mohamed, and M. Guizani, "Traffic analysis attacks on tor: A survey," in *2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT)*, 2020, pp. 183–188. [Online]. Available: https://css.csail.mit.edu/6.858/2023/readings/tor-traffic-analysis.pdf

[22] P. Mittal, A. Khurshid, J. Juen, M. Caesar, and N. Borisov, "Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011, p. 215–226. [Online]. Available: https://doi.org/10.1145/2046707.2046732

[23] S. Bagui, X. Fang, E. Kalaimannan, S. C. Bagui, and J. Sheehan, "Comparison of machine-learning algorithms for classification of vpn network traffic flow using time-related features," in *Journal of Cyber Security Technology*, vol. 1, no. 2, 2017, pp. 108–126. [Online]. Available: https://doi.org/10.1080/23742917.2017.1321891

[24] D. Price, "Sizing the piracy universe," in *London: NetNames*, 2013. [Online]. Available: http://www2.itif.org.s3.amazonaws.com/panelists+powerpoints/2013-netnames-piracy.pdf

[25] D. Ibosiola, B. Steer, A. Garcia-Recuero, G. Stringhini, S. Uhlig, and G. Tyson, "Movie pirates of the caribbean: Exploring illegal streaming cyberlockers," in *Proceedings of the International AAAI Conference on Web and Social Media*, 2018. [Online]. Available: https://www.eecs.qmul.ac.uk/~tysong/files/ICWSM18.pdf

[26] M. Z. Rafique, T. van Goethem, W. Joosen, C. Huygens, and N. Nikiforakis, "It's free for a reason: Exploring the ecosystem of free live streaming services," in *Network and Distributed System Security Symposium*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:14475991

[27] K. Rudman, M. Bonenfant, M. Celik, J. Daniel, J. Haitsma, and J.-P. Panis, "Toward real-time detection of forensic watermarks to combat piracy by live streaming," in *SMPTE 2014 Annual Technical Conference & Exhibition*, 2014, pp. 1–12. [Online]. Available: https://www.researchgate.net/publication/293640427_Toward_Real-Time_Detection_of_Forensic_Watermarks_to_Combat_Piracy_by_Live_Streaming

[28] L. Decker, D. Leite, L. Giommi, and D. Bonacorsi, "Real-time anomaly detection in data centers for log-based predictive maintenance using an evolving fuzzy-rule-based approach," in *2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, 2020, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9177762

[29] A. Kim, M. Park, and D. H. Lee, "Ai-ids: Application of deep learning to real-time web intrusion detection," in *IEEE Access*, vol. 8, 2020, pp. 70 245–70 261. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9063416

[30] W. N. H. Ibrahim, S. Anuar, A. Selamat, O. Krejcar, R. G. Crespo, E. E. Herrera-Viedma, and H. Fujita, "Multilayer framework for botnet detection using machine learning algorithms," in *IEEE Access*, vol. 9, 2021, pp. 48 753–48 768. [Online]. Available: https://api.semanticscholar.org/CorpusID:233138006

[31] H. Kanehara, Y. Murakami, J. Shimamura, T. Takahashi, D. Inoue, and N. Murata, "Real-time botnet detection using nonnegative tucker decomposition," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC)*, 2019, p. 1337–1344. [Online]. Available: https://doi.org/10.1145/3297280.3297415

[32] M. Gromov, D. Arnold, and J. Saniie, "Edge computing for real time botnet propagation detection," pp. 13–16, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:253518351

[33] A. Oussous, F.-Z. Benjelloun, A. Ait Lahcen, and S. Belfkih, "Big data technologies: A survey," in *Journal of King Saud University - Computer and Information Sciences*, vol. 30, no. 4, 2018, pp. 431–448. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157817300034

[34] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Commun. ACM*, vol. 51, no. 1, 2008, p. 107–113. [Online]. Available: https://doi.org/10.1145/1327452.1327492

[35] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. H. Zulkernine, and S. Khan, "A survey of distributed data stream processing frameworks," in *IEEE Access*, vol. 7, 2019, pp. 154 300–154 316. [Online]. Available: https://api.semanticscholar.org/CorpusID:204955699

[36] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013, p. 423–438. [Online]. Available: https://doi.org/10.1145/2517349.2522737

[37] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink : Stream and batch processing in a single engine," in *IEEE Data(base) Engineering Bulletin*, vol. 36, 2015, pp. 28–33. [Online]. Available: https://api.semanticscholar.org/CorpusID:263802939

[38] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu, "Ibm streams processing language: analyzing big data in

motion," in *IBM Journal of Research and Development*, vol. 57, no. 3–4, 2013. [Online]. Available: https://doi.org/10.1147/JRD.2013.2243535

[39] T. Jia, Y. Wu, C. Hou, and Y. Li, "Logflash: Real-time streaming anomaly detection and diagnosis from system logs for large-scale software systems," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 2021, pp. 80–90. [Online]. Available: https://api.semanticscholar.org/CorpusID:246753962

[40] L. Xu, S. Venkataraman, I. Gupta, L. Mai, and R. Potharaju, "Move fast and meet deadlines: Fine-grained real-time stream processing with cameo," in *Symposium on Networked Systems Design and Implementation*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:222177584

[41] A. S. Gomes, J. Oliveirinha, P. Cardoso, and P. Bizarro, "Railgun: managing large streaming windows under mad requirements," 2021. [Online]. Available: https://arxiv.org/abs/2106.12626

[42] S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl, "Analyzing efficient stream processing on modern hardware," in *Proceedings of the VLDB Endowment*, vol. 12, no. 5, 2019, p. 516–530. [Online]. Available: https://doi.org/10.14778/3303753.3303758

[43] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, and M. Andrews, "The memsql query optimizer: A modern optimizer for real-time analytics in a distributed database," in *Proceedings of the VLDB Endowment*, vol. 9, no. 13, 2016, p. 1401–1412. [Online]. Available: https://doi.org/10.14778/3007263.3007277

[44] J. Sun, Z. Shang, G. Li, D. Deng, and Z. Bao, "Balance-aware distributed string similarity-based query processing system," in *Proceedings of the VLDB Endowment*, vol. 12, no. 9, 2019, p. 961–974. [Online]. Available: https://doi.org/10.14778/3329772.3329774

[45] Z. Cai and T. Shi, "Distributed query processing in the edge-assisted iot data monitoring system," in *IEEE Internet of Things Journal*, vol. 8, no. 16, 2021, pp. 12 679–12 693. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9206577

[46] J. Kossmann, T. Papenbrock, and F. Naumann, "Data dependencies for query optimization: a survey," in *The VLDB Journal*, vol. 31, no. 1, 2021, p. 1–22. [Online]. Available: https://doi.org/10.1007/s00778-021-00676-3

[47] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," in *Proceedings of the VLDB Endowment*, vol. 12, no. 11, 2019, p. 1705–1718. [Online]. Available: https://doi.org/10.14778/3342263.3342644

[48] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, 2021, p. 1275–1288. [Online]. Available: https://doi.org/10.1145/3448016.3452838

[49] Y. Wang, L. Zhang, Y. Ren, and W. Zhang, "Nexus: Bringing efficient and scalable training to deep learning frameworks," in *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2017, pp. 12–21. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8107427

[50] I. Chebbi, N. Mellouli, I. R. Farah, and M. Lamolle, "Big remote sensing image classification based on deep learning extraction features and distributed spark frameworks," in *Big Data and Cognitive Computing*, vol. 5, no. 2, 2021. [Online]. Available: https://www.mdpi.com/2504-2289/5/2/21

[51] D. Syed, A. Zainab, A. Ghrayeb, S. S. Refaat, H. Abu-Rub, and O. Bouhali, "Smart grid big data analytics: Survey of technologies, techniques, and applications," in *IEEE Access*, vol. 9, 2021, pp. 59 564–59 585. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9272794

[52] S. Tang, B. He, C. Yu, Y. Li, and K. Li, "A survey on spark ecosystem: Big data processing infrastructure, machine learning, and applications," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 1, 2022, p. 71–91. [Online]. Available: https://doi.org/10.1109/TKDE.2020.2975652

[53] M. S. Wiewiórka, A. Messina, A. Pacholewska, S. Maffioletti, P. Gawrysiak, and M. J. Okoniewski, "Sparkseq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision," in *Bioinformatics*, vol. 30 18, 2014, pp. 2652–3. [Online]. Available: https://api.semanticscholar.org/CorpusID:9656129

[54] S. Hussain, J. Ferzund, and R. Ul-Haq, "Prediction of drug target sensitivity in cancer cell lines using apache spark," in *Journal of Computational Biology*, vol. 26, no. 8, 2019, pp. 882–889. [Online]. Available: https://doi.org/10.1089/cmb.2018.0102

[55] K. Dutta and M. Jayapal, "Big data analytics for real time systems," in *Big Data analytics seminar*, 2015, pp. 1–13. [Online]. Available: https://www.researchgate.net/profile/Kamalika-Dutta/publication/304078196_Big_Data_Analytics_for_Real_Time_Systems/links/576594e408aedbc345f38226/Big-Data-Analytics-for-Real-Time-Systems.pdf

[56] L. Li, H. Xiong, Z. Guo, J. Wang, and C. Xu, "Smartpc: Hierarchical pace control in real-time federated learning system," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 406–418. [Online]. Available: https://api.semanticscholar.org/CorpusID:203582658

[57] H. Zhang, J. Bosch, and H. H. Olsson, "Real-time end-to-end federated learning: An automotive case study," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2021, pp. 459–468. [Online]. Available: https://api.semanticscholar.org/CorpusID:232307438

[58] M. Nasr, A. Houmansadr, and A. Mazumdar, "Compressive traffic analysis: A new paradigm for scalable traffic analysis," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, p. 2053–2069. [Online]. Available: https://doi.org/10.1145/3133956.3134074

[59] K. Lee, J. Caverlee, and S. Webb, "Uncovering social spammers: social honeypots + machine learning," in *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2010, p. 435–442. [Online]. Available: https://doi.org/10.1145/1835449.1835522

[60] M. McCord and M. Chuah, "Spam detection on twitter using traditional classifiers," in *Proceedings of the 8th International Conference on Autonomic and Trusted Computing (ATC)*, 2011, p. 175–186. [Online]. Available: https://api.semanticscholar.org/CorpusID:15569859

[61] L. Das, L. Ahuja, and A. Pandey, "Analysis of twitter spam detection using machine learning approach," in *2022 3rd International Conference on Intelligent Engineering and Management (ICIEM)*, 2022, pp. 764–769. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9853100

[62] S. Sedhai and A. Sun, "Semi-supervised spam detection in twitter stream," in *IEEE Transactions on Computational Social Systems*, vol. 5, 2017, pp. 169–175. [Online]. Available: https://api.semanticscholar.org/CorpusID:206656944

[63] I. Bate and U. Khan, "Wcet analysis of modern processors using multi-criteria optimisation," *Empirical Software Engineering*, vol. 16, no. 1, p. 5–28, 2011. [Online]. Available: https://doi.org/10.1007/s10664-010-9133-9

[64] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution," in *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, 2006, pp. 57–66. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4032336

[65] A. Burns and S. Edgar, "Predicting computation time for advanced processor architectures," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, 2000, p. 89–96. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=853996

[66] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres, "Testing real-time systems using genetic algorithms," in *Software Quality Journal*, vol. 6, no. 2, 1997, p. 127–135. [Online]. Available: https://doi.org/10.1023/A:1018551716639

[67] S. Law and I. Bate, "Achieving appropriate test coverage for reliable measurement-based timing analysis," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, 2016, pp. 189–199. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7557880

[68] I. Bate, D. Griffin, and B. Lesage, "Establishing confidence and understanding uncertainty in real-time systems," in *Proceedings of the 28th International Conference on Real-Time Networks and Systems (RTNS)*, 2020, p. 67–77. [Online]. Available: https://doi.org/10.1145/3394810.3394816

[69] M. Búr, K. Marussy, B. H. Meyer, and D. Varró, "Worst-case execution time calculation for query-based monitors by witness generation," in *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 6, 2021. [Online]. Available: https://doi.org/10.1145/3471904

[70] P. Goonetilleke, A. Knorre, and A. Kuriksha, "Hydra: Lessons from the world's largest darknet market," in *Criminology & Public Policy*, vol. 22, no. 4, 2023, pp. 735–777. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/1745-9133.12647

[71] K. Oosthoek, M. Van Staalduinen, and G. Smaragdakis, "Quantifying dark web shops' illicit revenue," in *IEEE Access*, vol. 11, 2023, pp. 4794–4808. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10012354

[72] J. L. Garcia-Dorado, A. Finamore, M. Mellia, M. Meo, and M. Munafo, "Characterization of isp traffic: Trends, user habits, and access technology impact," in *IEEE Transactions on Network and Service Management*, vol. 9, no. 2, 2012, pp. 142–155. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6158423

[73] S. Burckhardt, A. Gotsman, and H. Yang, "Understanding eventual consistency," *Microsoft Research*, 2013. [Online]. Available: https://www.cs.ox.ac.uk/people/hongseok.yang/paper/pods13-submitted.pdf

[74] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a replicated logging system with apache kafka," in *Proceedings of the VLDB Endowment*, vol. 8, no. 12, 2015, p. 1654–1655. [Online]. Available: https://doi.org/10.14778/2824032.2824063

[75] H. Wu, Z. Shang, and K. Wolter, "Performance prediction for the apache kafka messaging system," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019, pp. 154–161. [Online]. Available: https://api.semanticscholar.org/CorpusID:203654847

[76] L. T. D. Paolis, V. D. Luca, and R. Paiano, "Sensor data collection and analytics with thingsboard and spark streaming," in *2018 IEEE Workshop on Environmental, Energy, and Structural Monitoring Systems (EESMS)*, 2018, pp. 1–6. [Online]. Available: https://api.semanticscholar.org/CorpusID:49652734

[77] B. Zhou, J. Li, X. Wang, Y. Gu, L. Xu, Y. Hu, and L. Zhu, "Online internet traffic monitoring system using spark streaming," in *Big Data Mining and Analytics*, vol. 1, no. 1, 2018, pp. 47–56. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8268735

[78] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[79] R. Shree, T. Choudhury, S. C. Gupta, and P. Kumar, "Kafka: The modern platform for data management and analysis in big data domain," in *2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*, 2017, pp. 1–5. [Online]. Available: https://api.semanticscholar.org/CorpusID:5090212

[80] A. Auradkar, C. Botev, S. Das, D. D. Maagd, A. Feinberg, P. Ganti, L. Gao, B. Ghosh, K. Gopalakrishna, B. Harris, J. Koshy, K. Krawez, J. Kreps, S.-H. Lu, S. Nagaraj, N. Narkhede, S. Pachev, I. Perisic, L. Qiao, T. Quiggle, J. Rao, B. Schulman, A. Sebastian, O. Seeliger, A. Silberstein, B. Shkolnik, C. Soman, R. Sumbaly, K. Surlaker, S. Topiwala, C. N. Tran, B. Varadarajan, J. Westerman, Z. White, D. Zhang, and J. Zhang, "Data infrastructure at linkedin," in *2012 IEEE 28th International Conference on Data Engineering*, 2012, pp. 1370–1381. [Online]. Available: https://api.semanticscholar.org/CorpusID:7453827

A

CorrForce configurations

A.1 Deployment over Docker

All components of CorrForce are designed for deployment using Docker containers. This section provides information about the Docker images and container deployment.

Deploying the Spark cluster begins with the creation of all necessary Spark components, including a Master node and multiple Worker nodes, each executed in separate containers. These containers use a custom Docker image based on *alpine:latest*, which includes the required Java version for running the compiled Spark code and the Spark tool package necessary for node deployment. Additionally, the HDFS nodes used for checkpointing are launched using the *apache/hadoop:3* image.

The Docker image for all Kafka-related nodes is *apache/kafka:latest*. This image not only automatically creates the broker service upon container launch, using configurations specified in Section A.2, but also ensures the latest version of Kafka is available.

Finally, for the ISP simulation, Docker containers are created based on custom Docker images for the client, OS, and ISP components. The client image is built on *ubuntu:20.04*, equipped with the Tor service and all necessary Python library dependencies required for the simulation. The OS image is based on the *debian:buster* distribution and includes necessary service dependencies such as *Tor* and *NGINX*, along with their configuration files. Finally, the ISP image is based on *ubuntu:20.04* and includes all required Python dependencies.

Table A.1: Broker and Controller configuration.

Environment Variable	Value
KAFKA_NODE_ID	ID_{Node}
KAFKA_PROCESS_ROLES	"broker,controller"
KAFKA_CONTROLLER_QUORUM_VOTERS	"1@(IP_1):9093,2@(IP_2):9093,3@(IP_3):9093"
KAFKA_LISTENER_SECURITY_PROTOCOL_MAP	"CONTROLLER:PLAINTEXT,BROKER:PLAINTEXT,OUTSIDE:PLAINTEXT"
KAFKA_LISTENERS	"BROKER://:9091,CONTROLLER://:9093,OUTSIDE://:9092"
KAFKA_INTER_BROKER_LISTENER_NAME	"BROKER"
KAFKA_ADVERTISED_LISTENERS	"OUTSIDE://(IP_{Host}):9092,BROKER://(IP_{Host}):9091"
KAFKA_CONTROLLER_LISTENER_NAMES	"CONTROLLER"
CLUSTER_ID	"4L6g3nShT-eMCtK–X86sw"
KAFKA_LOG_DIRS	"/tmp/kraft-combined-logs"
KAFKA_MESSAGE_MAX_BYTES	"20000000 (Bytes)"
KAFKA_MAX_MESSAGE_BYTES	"20000000 (Bytes)"
Replication Configurations	
KAFKA_DEFAULT_REPLICATION_FACTOR	"3"
KAFKA_MIN_INSYNC_REPLICAS	"2"
KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR	"3"
KAFKA_TRANSACTION_STATE_LOG_MIN_ISR	"2"
KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR	"3"
KAFKA_REPLICA_FETCH_MAX_BYTES	"20000000 (Bytes) "

A.2 DQS Kafka Cluster Configuration

Brokers are configured at launch through Kafka related environment variables. The set of adopted configurations for a cluster of three brokers is displayed in Table A.1. Among them are *KAFKA_PROCESS_ROLES*, defining the roles the broker plays, being either a simple broker, a controller or both. Through *KAFKA_CONTROLLER_-QUORUM_VOTERS*, any server in cluster, independent of its role, will be aware of the set of controllers that participate in quorum votes. *KAFKA_LISTENERS* sets what listeners the broker has active on which port, each listener being identified by its own name. Each broker will have three active listeners, one dedicated for inter broker communication(i.e. BROKER://:9091) as defined by *KAFKA_INTER_BROKER_LISTENER_NAME*, one for interaction between controllers(i.e. CONTROLLER://:9093) configured through *KAFKA_CONTROLLER_LISTENER_NAMES* and finally one for communication with producers and consumers(i.e. OUTSIDE://:9092), which also defined listed in *KAFKA_ADVERTISED_LISTENERS* to allow for external communications. Each of this listeners will be associated with a security protocol, set through *KAFKA_LISTENER_SECURITY_PROTOCOL_MAP*. As of now we do not focus on ensuring security in communication therefore we simply set the security protocol as PLAINTEXT. *KAFKA_LOG_DIRS* determines the directory where all topic messages will be stored within the server file system and *KAFKA_MESSAGE_MAX_BYTES*, *KAFKA_MAX_MESSAGE_BYTES* along with *KAFKA_REPLICA_FETCH_MAX_BYTES* will increase the possible message size producers can issue. To achieve the Kafka cluster guarantees presented in Section 5.4, *KAFKA_DEFAULT_REPLICATION_FACTOR*, *KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR* and *KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR* are set to three as we pretend to have partitions replicated across three replicas. To configure the dynamic set of ISRs kept by Kafka, *KAFKA_MIN_INSYNC_REPLICAS* and *KAFKA_TRANSACTION_STATE_LOG_MIN_ISR* defined to two, as per the requirement of tolerating one fault. These two configuration will only take effect when producers are configured with additional *ack* property set to 'all', hence requiring that at least a minimum of ISRs exists in order for the message to be considered committed.

Aside from the cluster, producers and consumers carry some final configurations. Apart from the bootstrap list and the Json serializer/de-serializer mentioned in Section 5.4, producers need to configure the request mechanism, as messages are sent to the brokers in requests. First we extend the request size by defining *max_request_size* as 200 000 000 bytes. Requests may contain multiple batches, one for each partition, containing multiple messages issued by the producer in between request transmissions. Smaller batch sizes will make the batching mechanism less effective and possibly reduce throughput, hence we set *batch_size* to 500 000 bytes opposed to the 16 384 bytes default value. We also define *linger_ms* to 50 ms as a request timeout, therefore a request is sent either if the maximum size of the batch was reached or timeout as expired. This solution is best compatible in heavy load scenarios, as the *linger_ms* configuration can affect the throughput if few messages are issued. Finally, consumers only require the *auto_offset_reset* option to be set to 'latest' in order to avoid consuming messages prior to the creation of the consumer.