Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Martin Ingesen

Real-Time Event Correlation for Windows Event Logs

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
June 2020

NTNU
Kunnskap for en bedre verden

Martin Ingesen

Real-Time Event Correlation for Windows Event Logs

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Abstract

New vulnerabilities and attack vectors are discovered every day. Cyber attacks can critically impact and cripple businesses that are targeted. Many of these cyber threats focus on penetrating the network of a business to steal valuable information, hold data as ransom or permanently destroy the business network. The cost of a cyber attack can be high, and is not only measured in lost data or equipment, but also the business reputation and client-base. This is why it is important to identify such attacks as soon as possible.

The most common way to do network security monitoring, is to use solutions that detect, alert and possibly prevent security incidents from occurring by monitoring the network traffic that flows to and from the computers in the business network, and out to the internet. But as businesses are moving to become more and more digital, and the workforce is getting accustomed to working from anywhere, be it from home, from the coffee shop or even from the beach, the business network-perimeter is slowly being eroded away.

The industry solution to this has been to shift focus away from network-based monitoring and detection, and shift the focus towards the endpoints in the network. Centralizing and analysing log data from multiple endpoints has become more and more commonplace in enterprises. Even though new technology has made it easier to collect and store huge amounts of events, the problem still persist on how to analyze and alert on those events in real time. There exist different solutions for correlating event logs, but we believe that the specialized software can be further enhanced to improve the performance of real time event correlation. In this thesis we propose an improved method for correlating Windows event logs in near real-time.

Sammendrag

Nye sårbarheter og angrepsvektor blir funnet hver dag. Cyberangrep kan kritisk skade og påvirke bedrifter som blir angrepet. Mange av disse truslene fokuserer på å penetrere nettverket til bedriften for å stjele verdifull informasjon, holde data som gissel eller permanent ødelegge bedriftsnettverket. Kostnaden av et cyberangrep kan være høy, og er ikke bare målt i tapt data eller utstyr, men også bedriftens omdømme og kunder. Dette er grunnen til at det er viktig å identifisere slike angrep så raskt som mulig.

Den mest vanlige måten å bedrive sikkerhetsmonitorering av et nettverk, er ved å bruke løsninger som detekterer, alarmerer og muligens forhindrer sikkerhetshendelser fra å inntreffe ved å overvåke nettverkstrafikken som flyter mellom maskinene i bedriftsnettverket, og ut på internett. Men når bedrifter stadig blir mer og mer digitale, og arbeidsstyrken blir mer vandt til å jobbe fra hvor som helst, enten det er fra hjemme, fra kaffesjappa eller fra stranden, så eroderes bedriftens nettverksperimeter sakte men sikkert bort.

Industriens løsning på dette problemet har vært å skifte fokus vekk fra nettverksbasert overvåkning og deteksjon, og skifte fokus mot endepunktene i nettverket. Sentralisering og analysering av loggdata fra flere endepunkt har blitt mer og mer vanlig i større bedrfiter. Selv om ny teknologi har gjort det enklere å samle og lagre store mengder med eventer, så er det fremdeles et problem hvordan man skal analysere og alarmere på de eventene i sanntid. Det finnes forskjellige løsninger for å korrelere event logger, men vi mener at den type spesialisert programvare kan bli ytterligere forbedret for å øke ytelsen ved sanntidskorrelering av event logger. I denne oppgaven presenterer vi en forbedret metode for å korrelere Windows event logger i nær sanntid.

Acknowledgment

Foremost, I would like to express my sincere gratitude to my supervisor, Ass. Prof. Geir Olav Dyrkolbotn for providing excellent guidance, assistance and support during this thesis. I especially appreciate how my supervisor has facilitated guidance for me as a distance student at NTNU.

Special thanks go to my employer BDO AS, and especially Ingunn Holte and Håkon Lønmo, for allowing me time to research, study and write my thesis while working full-time.

I highly appreciate all motivation and support from friends and family throughout my studies. I especially couldn't have done this without my partner Cecilie which has supported me through all the ups and downs along the way. I would also like to mention our son Arthur who never failed to cheer me up when I was stuck or met an obstacle during my writing of this thesis.

M.I.
02-06-2020

Contents

Figures

Tables

Code Listings

Acronyms

API Application Programming Interface. 8

AV Anti-Virus. 2

DDoS Distributed Denial-of-Service. 13

FSM Finite-state machine. 25

GPO Group Policy Object. 8

HIDS Host-based Intrusion Detection System. 2

IDS Intrusion Detection System. 1, 25

IPS Intrusion Prevention System. 1

JSON JavaScript Object Notation. 45

NSM Network Security Monitoring. 1

SEC Simple Event Correlator. 5, 7, 16, 25, 26, 28, 32, 35, 38–40, 42, 44, 45, 47–49, 55, 57, 61–63, 65, 66

SIEM Security Information and Event Management. 3, 4, 16, 32

SQL Structured Query Language. 28

Sysmon System Monitor. 5, 7, 10

XML Extensible Markup Language. 8

YAML YAML Ain't Markup Language. 32, 49

Chapter 1

Introduction

New vulnerabilities and attack vectors are discovered every day, and there is an increase in the development of new malware as shown in *The AV-TEST Security Report 2018/2019* by AV-TEST [1]. The report *M-Trends 2020* by FireEye Mandiant Services [2] underlines the fact cyber attacks can critically impact and cripple businesses that are targeted. Many of these cyber threats focus on penetrating the network of a business to steal valuable information, hold data as ransom or permanently destroy the business network. The cost of a cyber attack can be high, and is not only measured in lost data or equipment, but also the business reputation and client-base. This is why it is important to identify such attacks as soon as possible.

Traditionally, Network Security Monitoring (NSM) has been essential to avert these cyber threats and attacks. NSM is the collection, analysis, and escalation of indications and warnings to detect and respond to intrusions in the network. The goal is to detect and respond to threats as early as possible to prevent unauthorized access, misuse, destruction or data theft.

The most common way to do network security monitoring, is to use solutions known as Intrusion Detection System (IDS) or Intrusion Prevention System (IPS) as described by Liu *et al.* [3]. These systems are used to detect, alert and possibly prevent security incidents from occurring by monitoring the network traffic that flows to and from the computers in the business network, and out to the internet. The main benefits of using these network-based solutions, is that there is no need to alter the existing infrastructure or install any software on the hosts in the network. The solutions monitor everything on the network segment they are placed in, regardless of the operating systems (OS) running on the hosts. An additional factor has been the fact that these solutions have a lower cost of setup and maintenance than host-based solutions that require installing or configuring software on the hosts themselves.

But as businesses are moving to become more and more digital, and the workforce is getting accustomed to working from anywhere, be it from home, from the coffee shop or even from the beach, the business network-perimeter is slowly being eroded away. As of writing this, the COVID-19 virus is spreading across the globe,

and employees all around the world are forced to stay at home to reduce the risk of spreading the disease. This global pandemic is forcing those businesses who have not already adapted to a remote workforce, to introduce work-from-home quickly as described by Kramer and Kramer [4]. In addition to the work-from-home factor, we are also seeing a rise in encrypted traffic, both between hosts, but also out to the wider internet. Privacy-enhancing technologies like DNS-over-TLS/DNS-over-HTTPS, free TLS certificates and browsers marking unencrypted websites as "unsafe" are pushing the bar on moving to a fully-encrypted internet. Unless the business chooses to utilize TLS interception to "see" the encrypted traffic inline using their traditional network security monitoring solutions, they are increasingly becoming blind to the threats that might hide behind encrypted communications. There is also no visibility into what is actually happening on the hosts in the network, unless there is data transmitted across the network that can be analyzed. All of these factors contribute to a reduced value in network-based security monitoring.

The industry solution to this has been to shift focus away from network-based monitoring and detection, and shift the focus towards the endpoints in the network as said by Liu *et al.* [3]. The different solutions for endpoint protection have historically been hard to install, configure and maintain on the individual hosts in a business, and the alerts produced by the anti-virus or host monitoring software has to be transmitted and stored in a central location, as discussed in the work done by Brattstrom and Morreale [5]. In addition, performance degradation on the hosts caused by the resource-intensive software required for detection, prevention and transmitting alerts has been of concern.

First of all we have Host-based Intrusion Detection System (HIDS) which monitor the dynamic state of the host, and alerts on system changes that are out-of-place. This is usually based on a database containing the cryptographic hash of known-good files. The HIDS then monitor the files for any changes, and report any changes to a central location.

Then we have the common anti-virus/anti-malware/endpoint protection software. These software solutions usually contain a range of different detection and prevention methods, and usually incorporates a variety of signature-based, heuristic-based, data mining and machine learning detection. Commercial-grade Anti-Virus (AV) usually reports their findings to a central location for analysis. For anti-virus to protect its integrity and detect malice it has to run with high privileges on the host. Any vulnerabilities in the AV engine can then have fatal consequences allowing for instance privilege escalation on the host. There has been concerns regarding system instability caused by bugs in the AV engine or slow network connections caused by the AV doing network inspection. These faults are usually patched or corrected quickly by the vendor, but might still be of concern to the system administrators.

Lastly, we have event forwarding, which is software that sends the events generated by the OS to a central location for detection, analysis and forensic purposes. Storing all the logs, not just alerts like anti-virus and HIDS might do, in a central

location has the added benefit of being able to be searched in after-the-fact. This makes event forwarding very valuable for forensic purposes and for developing new detections based on historical data. Event forwarding requires knowledge of what logs to forward and what to filter out. The number of events that are generated per second can vary, and being able to estimate the amount of logs are important so that the central log collection can be scaled appropriately to accommodate the volume of logs that are being ingested and stored. In recent years, the technology both for configuring and maintaining software on the hosts and systems for ingesting host data to a central location has done great leaps. Vendors of security products have made their software simpler to configure, usually via a cloud-based console. Storage is in general cheaper, and Security Information and Event Management (SIEM) software has made it simpler to monitor and analyze large volumes of event and log data.

1.1 Problem description

Even though new technology has made it easier to collect and store huge amounts of events, the problem still persist on how to analyze and alert on those events in real time when collected centrally. A problem that occurs when companies are collecting more and more logs, is that actively hunting and alerting on badness in those logs are becoming harder and more complex as told by Fatemi and Ghorbani [6]. A single log item from a single source is not enough to properly analyze what has happened in a system. Only by cross-correlating several log lines and log sources are we able fully understand the situation at hand and create detection that are of high quality.
While modern SIEM software like Splunk [7], QRadar [8] and RSA NetWitness [9] support searching, analyzing and alerting in various degrees, quality SIEMs are usually heavyweight, expensive, licensed by how many gigabytes are ingested per day. The alert rules can be hard to create, manage and share between analysts, and probably the most significant factor is that the alerts are only generated after the log data has been indexed. This adds unnecessary latency when we optimally want near real-time alerting. Traditionally in a SIEM, logs are analyzed after-the-fact by an analyst. This is a major drawback, as this type of security monitoring is reactive and error-prone, and problems are only detected in hindsight as explained by Landauer *et al.* [10].
When considering free or open-source solutions like OSSIM [11], OSSEC [12] and SEC [13] to correlate event logs in real-time, they are often lacking in terms of performance and ease-of-use. In addition, when considering distributed company environments, the hosts are not always able to send their event logs at the same time. There will be delays based on the geographical location of the host, network latency or network connectivity issues. Events may be ingested in the "wrong" (non-sequential) order, or asynchronous with other hosts.

1.2 Justification, motivation and benefits

Today, event log correlation is usually done centrally using built-in functionality in a SIEM, or using specialized software that processes and correlates events before they are ingested into a central storage system. As the volume of ingested events increase, there is a big demand for solutions that are able to correlate large amounts of event log in near real time, while also addressing correlation-problems with regard to data latency, asynchronous events and time drift.
Each host generate a huge amount of events that can be available to us for analysis and correlation, and can give deep insight into what is happening on each system. While we have this goldmine of host event data, we can not simply apply signature-based alerting like we commonly see in anti-virus products. The reason for this is that it is much harder to tell if a single event contains malice. A event might for example contain the information that a specific user deleted a file. This could be malicious, or it could be benign. The context around that event decides if it is malicious activity or not. That level of context-awareness is impossible to get with regular signatures, and is why event correlation can be so powerful, but tricky. Another benefit of centrally analyzing event data from multiple hosts is the cross-host correlation that can be done. It makes it possible to create correlations that identify host-to-host interactions, lateral movement and attacker behaviour across the whole network, which previously only was possible with network-based monitoring. In the Microsoft Windows operating systems, those logs are known as Windows Event Logs.
Modern approaches in cyber security shift from a purely forensic to a proactive analysis of event logs as told by He *et al.* [14]. We believe that the specialized software can be further enhanced to improve the performance of real time event correlation. In this thesis we contribute an improved method for correlating Windows Event logs in near real-time, while at the same time taking care to address the problems with might occur with log ingestion delays and asynchronous events.

1.3 Research questions

To address the problems outlined in 1.1, the following research questions have been developed:

Hypothesis: We believe that we are able to improve upon current research and methods for real time event correlation, by utilizing a compiled, multi-threaded programming language and better rule formats.

Research questions:

1. What is the state of the art for real time event correlation?
2. How can we improve the way real time event correlation is done for Windows Event Logs?
3. What is the performance of our proposed method, and how does it compare to other methods?

1.4 Planned contributions

The primary contribution of this project is an improved method for correlating Windows Event Logs in time, in near real time. The goal of this thesis is to explore ways to improve real time log correlation both performance-wise but also addressing the problems that occur when analyzing asynchronous events or when experiencing log ingestion delays.

1.5 Thesis outline

This section presents an overview of the thesis and a short summary of each chapter.

Chapter 2: Background
First of all we give a give an introduction to event logs, Windows Event logs and System Monitor (Sysmon). We will take a look at the field of event correlation, and highlight some of the relevant techniques for correlating events. We then discuss Simple Event Correlator (SEC), and various types of rules that can be used with rule-based event correlation.

Chapter 3: Methodology
In this chapter we outline the methodology and steps we will take to address our research questions. First we look at how we can improve how real time event correlation is done, and afterwards we discuss how we can measure the performance of our solution.

Chapter 4: Experiments
Here we introduce our improved implementation. We outline the software and hardware specifications used, the dataset collection and required preprocessing is presented, and we introduce our solution in two steps.

Chapter 5: Results
In this chapter we present the results from our experiments, both looking at the datasets used, and measuring the performance of our implementations.

Chapter 6: Discussion
Here we discuss our findings in more detail, looking at the bigger picture. We also outline any future work.

Chapter 7: Conclusion
Finally we conclude by tying all ends together in a final summary of our thesis.

Chapter 2

Background

In this chapter we will give an introduction to event logs, and further elaborate on Windows Event logs and System Monitor (Sysmon). Then we will take a dive into the field of event correlation, and highlight some of the relevant techniques for correlating events, answering our first research question of what the state of the art for real time event correlation is. Furthermore we will take a look at Simple Event Correlator (SEC), as that is the rule-based event correlator that we will focus on in this thesis. Finally we will take a look at various types of rules that can be used with rule-based event correlation.

2.1 Event logs

In general terms, a event is *something* that happened at a point in time. It could be anything, like a bank transaction, a user logging in to a system, the fire alarm being pulled, that your food delivery has arrived, and so forth. In regards to computers, events are something that happens on the individual computer systems. There can be events for a broad range of use cases like events related to system components, such as drivers and built-in interface elements, events related to programs installed on the system or events related to security, such as logon attempts and resource access.
The original reason why these logs are kept is such that system administrators can use them to debug software or configuration issues. In recent years, security professionals have started reviewing and using these logs as a mean to analyze and detect what has happened on a system. The event logs can give the people doing digital forensics valuable insight into a machine compromise, or help detect malicious activity as it is happening. Historically, the event logs has purely been used as a reactive log source, and only with recent shifts has been getting more focus as explained by He *et al.* [14].
The amount of events that are logged on a machine varies greatly depending on how it is configured and what the software installed on the system choose to log. Depending on the system, event logs might have to be manually enabled or configured to provide the valuable insight into the events of the system. In

addition, there is no standardized way that logs are created. While there exist various attempts at creating a standard like Common Event Format (CEF)[15], Log Event Extended Format (LEEF)[16], Common Information Model (CIM)[17] and Intrusion Detection Message Exchange Format (IDMEF)[18], none of them have caught on. As outlined by He *et al.* [19] in the paper 'Towards Automated Log Parsing for Large-Scale Log Data Analysis', logs are generally unstructured, and analysing the logs relies on labor-intensive and error-prone manual inspection.
Automated log analysis and log mining has been discussed in various ways before (Xu *et al.* [20], Fu *et al.* [21], He *et al.* [22], Beschastnikh *et al.* [23], Shang *et al.* [24], Yuan *et al.* [25], Nagaraj *et al.* [26], Oprea *et al.* [27], and Gu *et al.* [28]) and will not be further covered here. Our focus for this thesis will be on Windows Event logs, and we will elaborate on that in Section 2.1.1. Support for other log formats is considered future work.

2.1.1 Windows Event Logs

Windows Event Log is a built-in capability of the Microsoft Windows operating systems.
According to Ultimate Windows Security [29], there are more than 400 different types of events that can be logged. Some of these event types have to explicitly be enabled, and some are enabled by default. As an example, if we want Windows to log events for when a network share object was accessed/added/modified/deleted, we have to enable that using Group Policy Object (GPO). The path for doing so can be found using the Group Policy Management Console and by navigating to "Computer Configuration -> Policies -> Windows Settings -> Security Settings -> Advanced Audit Policy Configuration -> Audit Policies -> Object Access -> Audit File Share" as seen in Figure 2.1.
Since the events are so verbose and plentiful, they can also overlap quite a lot. For instance when a new account is created, the event "4720: A user account was created" is created, as well as the events "4722: A user account was enabled.", "4724: An attempt was made to reset an account's password" and "4738: A user account was changed". This is shown in Figure 2.2.
In enterprise networks that utilize Active Directory for managing multiple hosts, these type of GPO settings can be configured centrally and applied to relevant machines. The above-mentioned file share events would for instance be interesting to enable for file servers, but not for other servers or client machines. If the enterprise uses some sort of central log collection, it is therefore necessary to configure and tune which events are saved, as that will affect how many events are sent over the wire and stored centrally.
When it comes to forwarding events and storing them, Windows Event logs are not stored in plain text on the system, but in a proprietary binary format as explained by Schuster [30]. To access the events programmatically, one have to go through the Windows Event Log Application Programming Interface (API)[31]. From the API it is possible to access the raw XML of the events. It is also possible to view

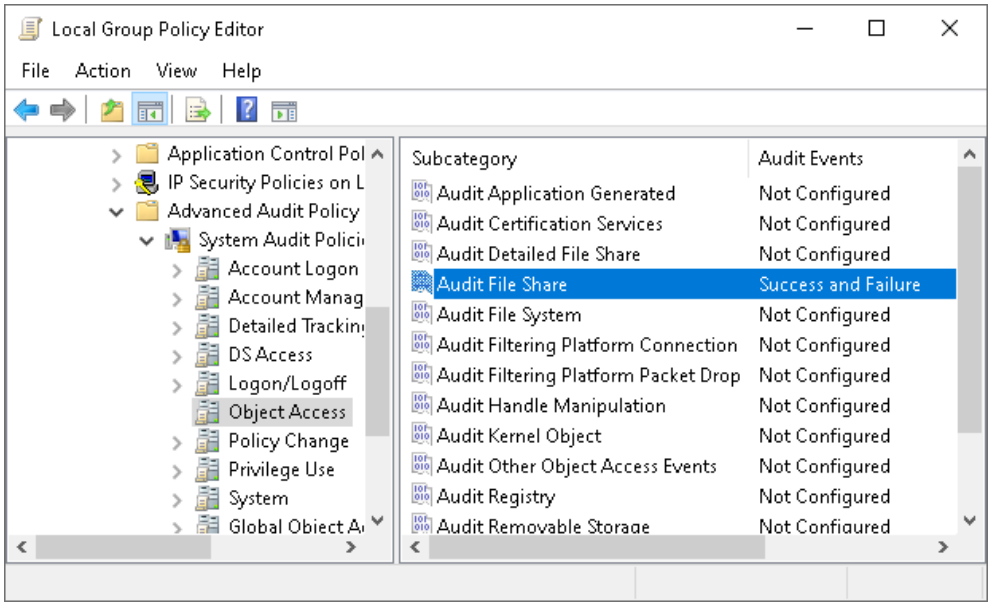


Figure 2.1: Screenshot of Local Group Policy Editor enabling file share auditing

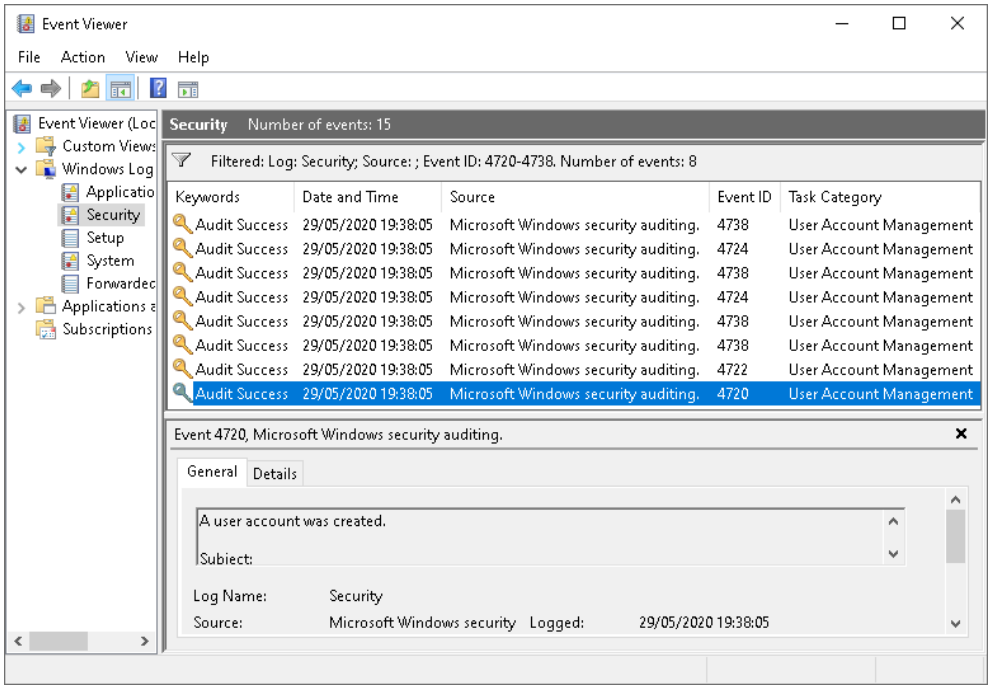


Figure 2.2: Screenshot of events related to user creation

the events in the built-in Event Viewer as seen in Figure 2.3. This is a program that allows for searching, filtering and viewing events. Each event contains a lot of information, and it is possible to view more details about each event as seen in Figure 2.4.

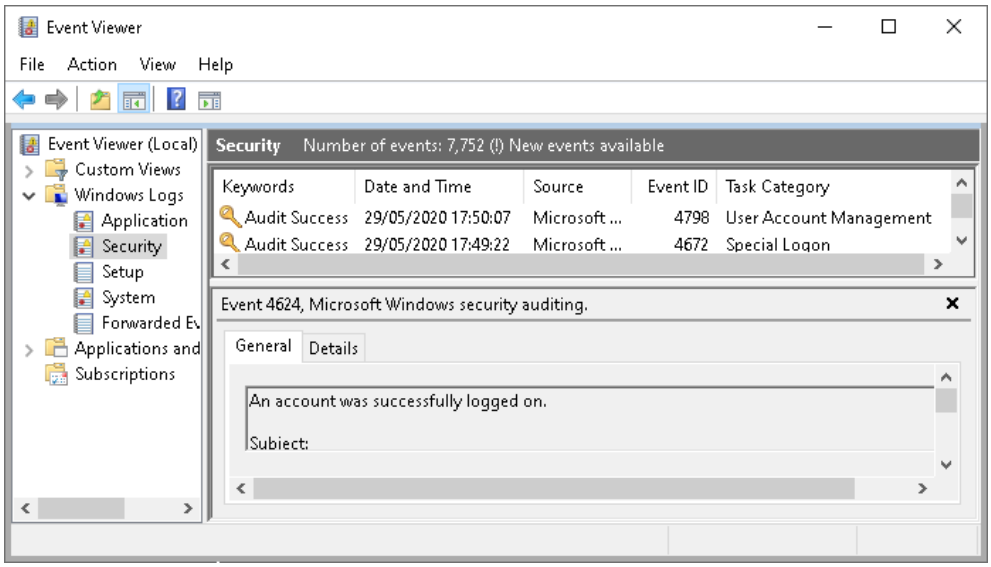


Figure 2.3: Screenshot of Event Viewer

In enterprises, Windows Event Logs are usually sent to a centralized location for storage and analysis, either using the built-in option called Windows Event Forwarding[32] or using custom agents like Splunk Universal Forwarder[33], Winlogbeat [34] or NXLog [35] to name a few.

Sysmon

System Monitor (Sysmon)[36] is an extension to the stock Windows Event Logs that allows for a more powerful customization of what events go into the event log. Using a kernel driver, Sysmon is able to add support for a wider variety of interesting events. The table 2.1 is a list of each event type that Sysmon can generate. Sysmon events do not replace those of regular Windows events, but creates events that contain detailed information about process creations, network connections, and changes to file creation time which can be used to help identify malicious or anomalous activity and understand how intruders and malware operate on your network.
For our experiments in this thesis, we will focus our attention towards the Sysmon process creation event (event ID 1). This event contains all the information necessary to detect which processes ran on a system, what its parent process was, what the command line arguments passed to the process was, and so forth.

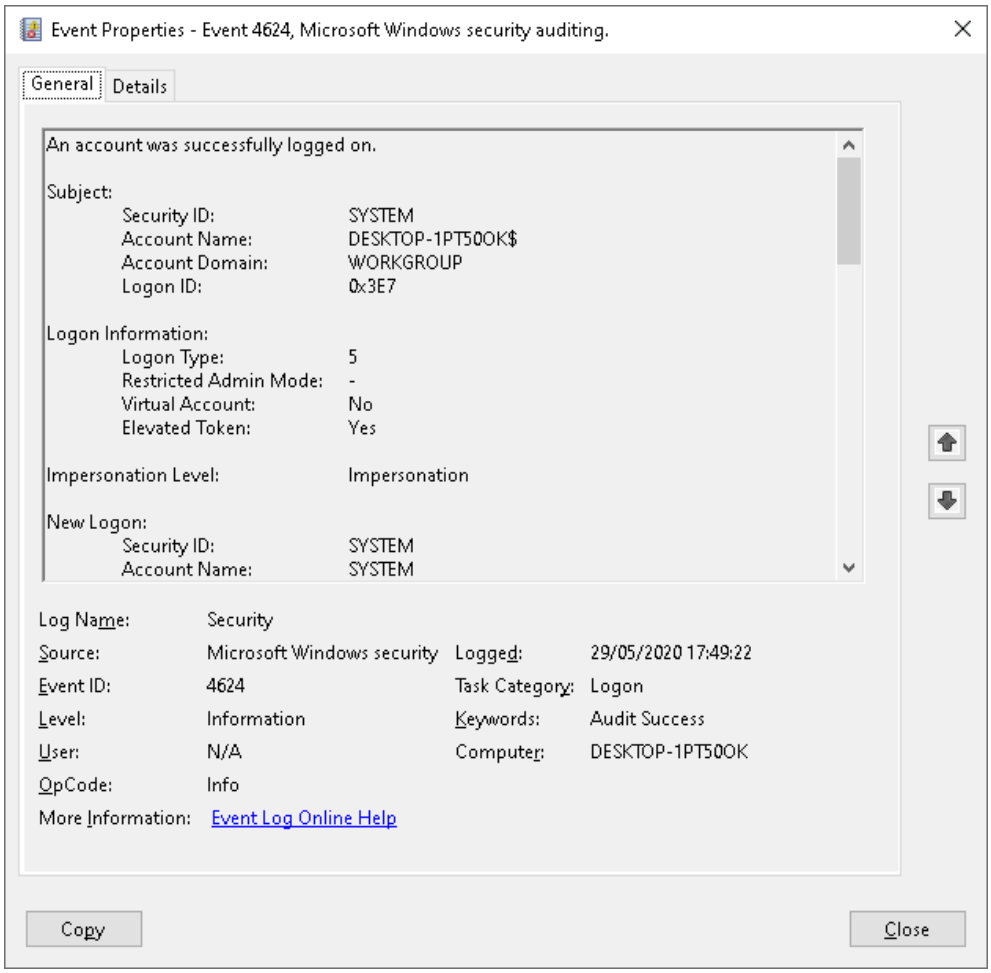


Figure 2.4: Screenshot of Event Properties

ID	Description
1	Process creation
2	A process changed a file creation time
3	Network connection
4	Sysmon service state changed
5	Process terminated
6	Driver loaded
7	Image loaded
8	CreateRemoteThread
9	RawAccessRead
10	ProcessAccess
11	FileCreate
12	RegistryEvent (Object create and delete)
13	RegistryEvent (Value Set)
14	RegistryEvent (Key and Value Rename)
15	FileCreateStreamHash
17	PipeEvent (Pipe Created)
18	PipeEvent (Pipe Connected)
19	WmiEvent (WmiEventFilter activity detected)
20	WmiEvent (WmiEventConsumer activity detected)
21	WmiEvent (WmiEventConsumerToFilter activity detected)
22	DNSEvent (DNS query)
255	Error

Table 2.1: List of Sysmon event types

2.2 Event correlation

As stated in Section 2.1, a event is something that happens at a point in time. Event correlation is a statistical relationship between random events that are not necessarily expressed by a rigorous functional relationship as stated by Prokhorov [37]. This means that the relationship between two events is based on the fact that the conditional probability of one of the events occurring, given the occurrence of another event, is different from the unconditional probability. There exists numerous ways to determine the dependency between two events, like Pearson coefficient according to Kent State University [38], Spearman's rank correlation coefficient as illustrated by Prokhorov [39], Kendall rank correlation coefficient as described in Prokhorov [40], Goodman and Kruskal's gamma by Goodman and Kruskal [41] just to name a few.
Event correlation is usually applied when we want to create a higher level of understanding, based on the information found in the events. By correlating events, we can gather up smaller events that in and of them self are not worthy an alarm, and create an over-arching alarm that encompasses the smaller events. Event correlation can be used for a wide range of cases, like root-cause analysis, fault detection and future prediction and its usage can be found in areas such as market and stock trends, fraud detection, system log analysis, network management and fault analysis, medical diagnosis and treatment, et cetera. In the information security sphere, correlation can be used for things like detecting patterns of Distributed Denial-of-Service (DDoS) attacks as shown by Wei *et al.* [42] and identifying subsets of data attributes for intrusion detection as outlined by Jiang and Cybenko [43] and for detection of attacks based on the relationships between network events as shown in Kruegel *et al.* [44].
Event correlation is a broad topic, and a complete overview is outside the scope of this thesis. The following sections will highlight some of the more popular event correlation methods, and particularly rule-based event correlation which will be the main focus for our thesis with regard to event correlation techniques.

2.2.1 Finite State Machines

A finite-state machine, a system is abstracted into mathematical model which can have exactly one of a finite number of states at a time. A finite-state machine has a fixed set of possible states, a set of inputs that change the state, and a set of possible outputs as described by Keller [45]. The next state of a finite-state machine is based on the current state that the machine is in, and the input that change the state. There are generally considered to be two kinds of finite-state machines, deterministic finite-state machines and non-deterministic finite-state machines. In a deterministic finite-state machine, every state has only one transition per input, as opposed to the non-deterministic state machine, where an input can lead to none, one or many transitions for a given state. Since the deterministic finite-state machine is a more strict version of the non-deterministic finite-state ma-

chine, that leads to that by definition, a deterministic finite-state machine is also a non-deterministic finite-state machine. For example, assuming that we have the following three events in order:

1. the process '$word.exe$' started
2. the process '$googlechrome.exe$' started
3. The process '$powershell.exe$' started

If we want to trigger an alert when we see the $word.exe$ process is created, and then the $powershell.exe$ process afterwards, we can design a simple non-deterministic state machine like the one in Figure 2.5. When applying the above-mentioned events to this finite state-machine, event number one will move our state from s_0 to s_1. Event number two will not do any transitions and change the state (one of the benefits of using a non-deterministic state machine). When event number three occurs, the state machine transitions from s_1 to s_2, and our accepting state is reached, which fulfills the state machine and we can create an alarm. One of the benefits of the finite-state model is that it is possible to specify if the

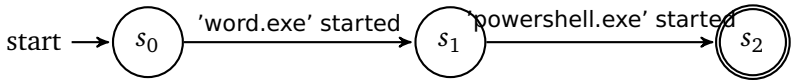


Figure 2.5: Example of non-deterministic finite-state machine

order of the events are important or not. If the event order is not of interested, a finite-state machine as shown in Figure 2.6 can represent the same case as seen in Figure 2.5.

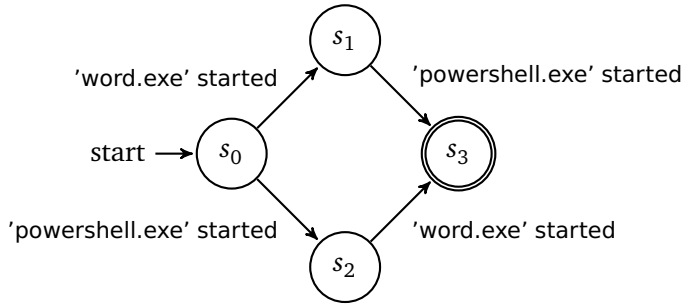


Figure 2.6: Example of non-deterministic finite-state machine

An approach to use finite-state machines for event correlation has been shown by Bouloutas *et al.* [46]. The authors use observed events that are generated by the monitored process to feed into the modelled finite-state machine that represent the monitored process. If an event arrives that leads to an invalid state in the model, an error is produced.
One of the main drawback with the finite-state machine is the missing notion of time. As shown in Figure 2.6 we can take into account order of events, but a finite-state machine does not separate on the time difference between events that are streamed into the model.

2.2.2 Rule-based Event Correlation

Rule-based event correlation software is historically known as a expert system. Expert systems is defined by Cronk *et al.* [47] as a "problem-solving software that embodies specialized knowledge in a narrow task domain to do work usually performed by a trained, skilled human.". According to Cronk *et al.* [47], expert systems are organized around three levels; data, control and knowledge. As shown in Figure 2.7, the data level is the working memory of the expert system that contains the events that are being processed. Then the knowledge level is the rule repository that contains the domain-specific expert knowledge. Finally we have the control level which consist of the inference engine that determines how to apply the rules from the knowledge base against the working memory.

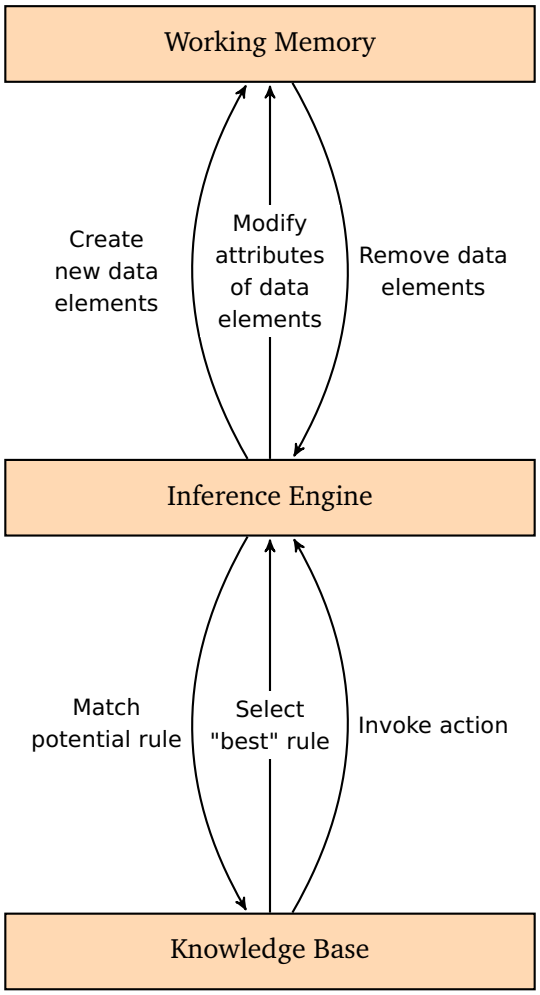


Figure 2.7: Model of rule-based expert systems

Traditionally, creating the rules that goes into a knowledge base is defined as two-fold; first you have the subject-matter expert which has the expertise and know-

ledge about which events you are interested in creating correlations against, and secondly the knowledge engineer which is familiar with how the expert system works and how the rules has to be written to be understood by the system. In more modern settings, usually the subject-matter expert and the knowledge engineer is the same person. This person has both the knowledge of which events are of interest, and the capability to implement, monitor and tune the rules necessary to detect the events that are of interest.
The value of a rule-based approach is that the rules in the knowledge base can be written with a close similarity to the human language. For example if we want to write a rule for the occurrence of two different events X and Y, it could be spelled out like "***IF*** *event X* ***AND*** *event Y* ***THEN*** *doAction*". This also makes it easier to deduce how and why an alert was triggered. We will take a further look at different rules that can be utilized with rule-based event correlation in Section 2.4.
In larger production environments, it is also important that the rules are specific enough, so that they do not generate too many alarms. There can be multiple reasons that a rule will trigger too many times. If the subject-matter expert is not specific enough when defining which conditions are to be added to the rule, or there can be a lack of proper events to analyze, such that to catch the behaviour that the subject-matter expert wants to detect, the knowledge engineer will have to write a more generic rule than wanted. Regardless, the knowledge engineer will have to tune the rule such that it will not flood the analysts with new alerts. Commonly such rules are ran in a test system with production input such that the knowledge engineer can collect metrics on how often the rules trigger alerts before adding the rule to production.
One of the main drawbacks, and probably the biggest reason for other types of event correlation is the lack of learning or adaptability, which menas that the same correlation will be made for every similar case every time as stated by Meira [48]. Networks may differ, so it is not given that a rule that fits into one network, can automatically be used in another. As outlined by Lewis [49], rule-based correlation tend to fail when presented with new or unexpected situations. In addition, creating new rules, maintaining old rules and adapting the rules in the knowledge base can be time-consuming. Regardless of these drawbacks, we see a common trend that rule-based systems are the most common when it comes to network-based monitoring (see Suricata [50], Snort [51]) as well as for log data in SIEMs like Splunk [7], OSSEC [12] and OSSIM [11].
There exists several different types of software that makes it possible to correlate events in real-time based on log data. From more simple projects like swatchdog [52], LogSurfer [53] and SEC [54], to more complex projects with multiple moving parts like Prelude [55], OSSEC [12], Wazuh [56], Apache Metron [57], MozDef [58], OpenNMS [59], OSSIM [11].
Throughout most of the literature regarding event correlation of log data, Simple Event Correlator (SEC) [54] has stuck out as one of the most popular software for doing event correlation on log data, as seen in Kont *et al.* [60], Farshchi [61] and Vaarandi [62] just to name a few. I will address SEC further under Section 2.3.

2.2.3 Case-based Reasoning

In case-based reasoning, a previously experienced problem and its solution is called a case. Case-based reasoning is based on the assumption that we can find a solution for a new problem by finding past cases that are similar, and then reusing the solution to solve the new problem. The reasoning is then further enforced by adding the problem and the solution to the case library for future use as described by Aamodt and Plaza [63]. As stated by Slade [64], case-based reasoning is similar to how humans approach new problems by assimilating past experiences and adapting them to new situations.
Figure 2.8 describes the cycle used in case-based reasoning from a high-level perspective. Under each step in the cycle there are multiple tasks that may be necessary to conduct before continuing on with the cycle. For instance, the "Retrieve" step might need to identify which features of the problem to search the Case Library for.

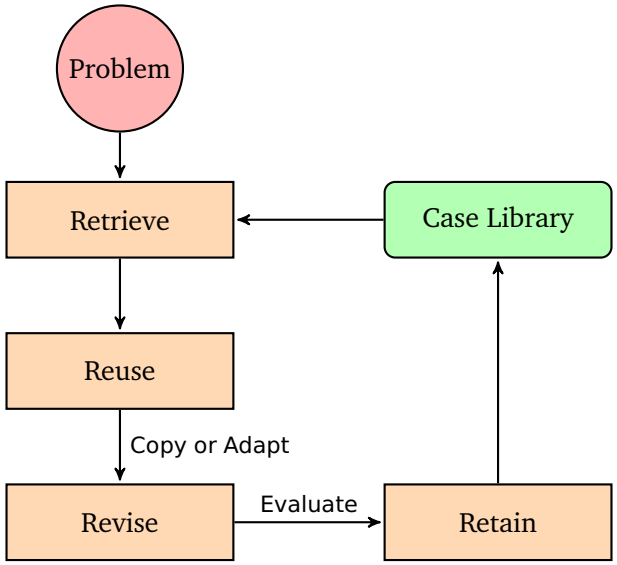


Figure 2.8: Case-based reasoning cycle

A example where this might be useful is in a Security Operations Center (SOC). A SOC receives a high number of alerts that have to be handled by an analyst to analyse and propose a response to the alert. The response can vary from simply suppressing the alert as a false-positive, sending an e-mail to the client to alert them, or escalating the alarm to the Incident Response team. Case-based reasoning can then be applied to new alerts by first retrieving the most similar alerts previously handled. The information stored in the previous case can then be used to handle the analysis or solution to an alert. The analyst will then revise the proposed solution, and retain the parts that might be useful for resolving similar future alerts. This follows the case-based reasoning cycle proposed in ‘Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches’

by Aamodt and Plaza [63].
The retrieval step is difficult because we need to find similar cases that offer solutions that are relevant. Cases may contain attributes that are irrelevant, which might not be clear to the automated retrieval process. An example of this could be the following: Consider that we receive an alert that a malicious file has been detected on a system. We get the IP, hostname, filename and hash of the file as part of the alert. The analyst decides that the file is benign through analysis. This is then stored as a case. If we then receive another similar alert containing yet again an IP, hostname, filename and hash of the file. The filename in this new alert is identical to the one we received earlier, but the hash is different. Using this data, the case-based reasoning engine should not propose a solution based the fact that the filenames are identical, since the file hashes are different, suggesting that the files are not the same. To solve this, both the work by Lewis [49] and Davies and Russell [65] propose creating "determination rules" or "determinators" that are either compound attributes or a pointer to which attributes to look at in the case. Additionally, adaptation of the old solutions to the new problem is a difficult task. While manual specification of the solution in the "Revise" step is possible and somewhat required, too much emphasis on manual intervention or adjustments will defeat the purpose of case-based reasoning. This is why according to Leake and Remindings [66] many case-based reasoning systems have adapted the cycle from Retrieve-Reuse-Revise-Retain to a much shorter Retrieve-Propose cycle that completely eliminates the adaptation.
In the paper 'A case-based approach to network intrusion detection', the authors Schwartz *et al.* [67] used the intrusion detection system Snort as a basis for a new case-based reasoning IDS that uses the Snort rule base as a case library. Snort rules may in general be too specific and fail to detect certain kind of intrusions, but with the case-based reasoning approach, the retrieval step in the cycle will take care of this by finding cases (rules) that are applicable to the network packet even though the vanilla rule would not create an alert on that packet. Kapetanakis *et al.* [68] argue that with the digital traces left by an attacker, it is possible to build a profile for that attacker which can be used to assist in future attacks to identify which attacker is attacking. In the paper written by Han *et al.* [69], the authors implemented a system called "WHAP" which uses case-based reasoning to compare cyber attacks against websites. WHAP builds on a large database of website defacements, which are custom webpages left on the victim server by the attacker to claim credit for a website hack. The system is then able to take new hacked websites as input, and output similar previous cases where it is likely that the website has been hacked by the same attacker. This can be useful for attribution and forensic investigations.

2.2.4 Model-based Reasoning

Model-based reasoning is a expert system where the target is to create a model that can be used to predict the outcome of input event or faults in the system.

The idea of modelling the structure and behavior of a system has its roots in the work done by Davies and Russell [65] where they explore the use of such models in troubleshooting digital electronics. There are no fixed way for how a system can or should be modelled. The model itself can be created as a logical formalization using pure mathematics, or as a simulated system using for example a game engine. As Dodig-Crnkovic and Cicchetti [70] highlight in their paper 'Computational Aspects of Model-Based Reasoning', there is an increased interest in automating the creation of the model of a system. This is based on the fact that creating and keeping a model consistent with the system it is supposed to model, is hard. Jakobson and Weissman [71] discuss model-based reasoning for alarm correlation for fault management in telecommunications networks in their paper 'Alarm correlation'.
In 'System Modeling and Diagnostics for Liquefying-Fuel Hybrid Rockets' written by Poll *et al.* [72], a figure similar to 2.9 is shown. It outlines the process for checking if a modelled system is consistent with the real world system it is supposed to replicate.

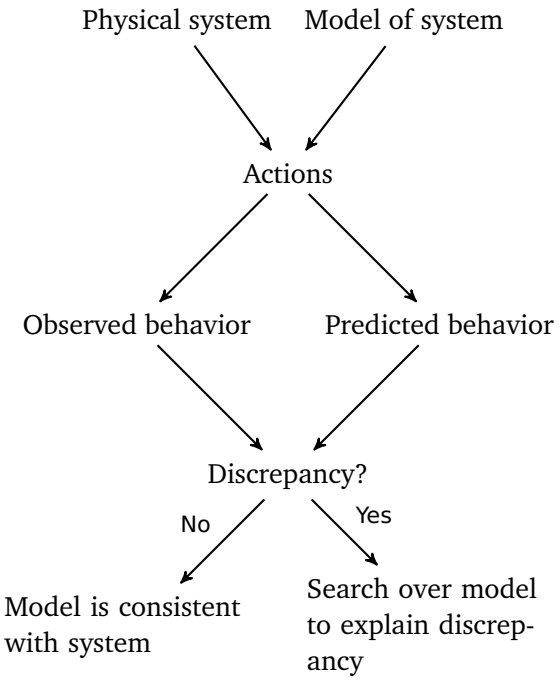


Figure 2.9: Illustration of model-based reasoning

As stated by Steinder and Sethi [73], one of the primary drawbacks of model-based reasoning is the requirement to have a well structured system to model and to keep that model updated. Systems that contain fluctuating objects like for example computer networks or network services are not trivial to represent in a formal model. More applicable areas might include hardware diagnostics like shown in the work by Davies and Russell [65], or other areas where it is possible

to model a more static target system, like for example automobile diagnostics. Finally, in 'A review of process fault detection and diagnosis: Part I: Quantitative model-based methods' by Venkatasubramanian *et al.* [74], they discuss that various implementations of model-based reasoning is quite computational complex, depending on number of objects in the model and their various inputs and outputs.

2.2.5 Codebook-based Event Correlation

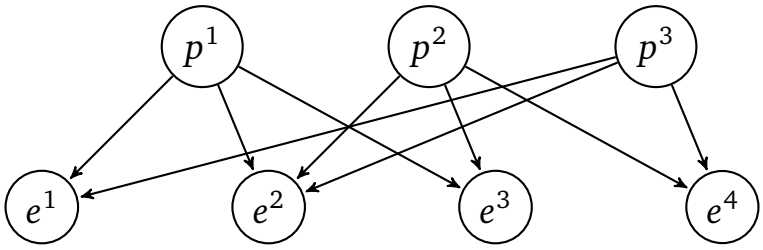


Figure 2.10: Example causality graph used for codebook-based event correlation

Yemini *et al.* [75] propose that the events caused by problems can be modelled as seen in figure 2.10 where the directed edges of the graph describe the causality of an event. p^x denotes a problem, and e^x denotes an event. To utilize the codebook, each problem node in the graph is converted into a binary vector that can be used to describe its relation to the events on the graph. This is known as a "code". The binary vector contains bits that corresponds to each event in the graph. If a bit is set to a 1, it indicates that the given problem causes the event that the bit corresponds to. A bit of 0 indicates that it does not cause the event. These codes then go into the codebook. If we convert the graph in figure 2.10 into a codebook, it will look like table 2.2. The graph and codebook needs to be sufficiently large to be able to identify all the problems. If the codebook is too small, it may omit events that are of interest to us. If the codebook is too large, it may contain events that are unnecessarily redundant. One way to approach the problem with codebooks that are too large, is to do what Yemini *et al.* [75] calls "codebook reduction". Codebook reduction is the process of removing events that are "universal" for all problems. In the figure 2.10 and the corresponding table 2.2 we can see that event e^2 is a common event for all the problems. Because of this redundancy, it can be remove to simplify the codebook as show in table 2.3. Further work has been done to enhance the efficiency of the codebook. Gupta and Subramanian [76] proposes a two step preprocessing algorithm that ensures mathematical provable codebooks and eliminates events that are unable to distinguish between problems.
When new events occur, the events are converted into a new binary vector. This vector is then compared with the codes in the codebook, and the code that is the most similar is chosen as a means to identify the problem. A simple approach for comparing the binary vectors could be a 1-to-1 comparison to see if the new binary vector exactly matches any of the codes in the codebook, but Yemini *et*

al. [75] propose to instead use Hamming distance to calculate the closest match. Using Hamming distance has several benefits, first of all it increases the tolerance for noise or lost events, secondly instead of choosing a single best candidate problem, we can defined a radius that will give us a codebook subset containing possible codes within the given Hamming distance radius. Because of the novel preprocessing down to binary vectors, codebook-based correlation is faster than other rule-based event correlation techniques. One of the more time-consuming tasks with regard to codebook-based event correlation is the creation of the problems and their mapping to symptom events. The most likely way to produce these codebooks will be as an expert system where a person with deep knowledge about the events in the system are able to map symptoms to problems. In addition, the process of selecting which events might be symptoms of a problem is similar to feature selection in the machine learning landscape. Feature selection is the process of selecting a subset of features that can be used in model construction, which is similar to how the codebook is generated.
One of the biggest limitations regarding codebook-based event correlation is that there is no built-in way to handle time. When a problem has been identified based on a number of symptoms, there is no time window applied, and there is no notion of event order. Furthermore the events do not contain any properties, and would require significant extending to take into account e.g. source hostname, username.

	e^1	e^2	e^3	e^4
p^1	1	1	1	0
p^2	0	1	1	1
p^3	1	1	0	1

Table 2.2: Codebook correlation matrix

	e^1	e^3	e^4
p^1	1	1	0
p^2	0	1	1
p^3	1	0	1

Table 2.3: Reduced codebook correlation matrix

2.2.6 Dependency Graphs

Similar to the dependency graph used in 2.2.5, Gruschke [77] suggests that a dependency graph can contain enough information to be used for event correlation, while also being simple to automatically generate. The dependency graph is a directed graph that maps the relationship managed objects. These objects can be hosts in a network, dependencies between software dependencies, and so forth. In figure 2.11 we have mapped a series of objects as an example. Events are mapped to their corresponding object in the graph (colored in blue, object b,

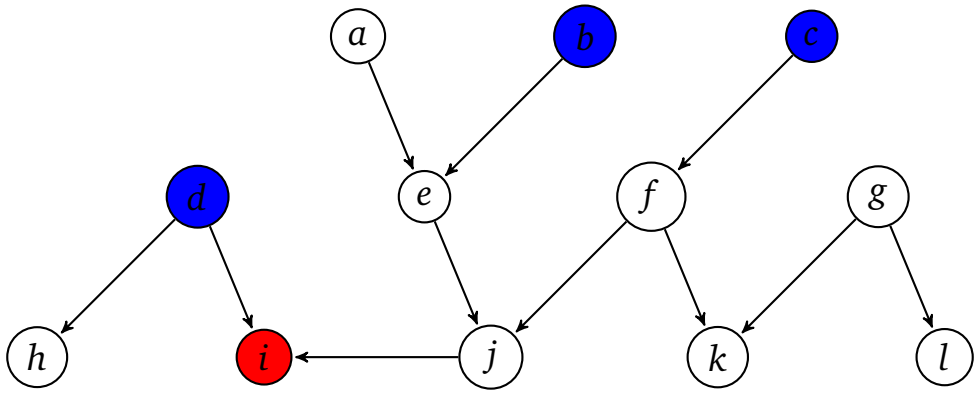


Figure 2.11: Example dependency graph

c and d). Then we walk the graph from those objects. As explained by Gruschke [77], when we optimally find one object node that are common for all the given events, we have most likely found the responsible node. In the example this is marked as red, object i. Gruschke [77] further outlines that the quality of the root-cause detection can be measured by the depth and length we need to walk the graph at. Objects that are further away from the initial object are less likely to be the root cause, and vica versa. One of the main drawbacks of dependency-graph-based correlation is the fact that it does not handle multiple, non-related problems very well. Gruschke [77] assumes that only one problem occurs at a time. If multiple problems occur that are not related or affect each other, finding the root-case may prove to be impossible, or select the wrong root-cause object. Assumes the events are for a single fault. Meaning it will not be able to handle detecting multiple failing nodes. As with the codebook-based event correlation we discussed in 2.2.5, dependency graphs also lack the notion of time. Additionally the dependency graph is not taking advantage of attributes on the nodes to further enhance the graph.

2.2.7 Bayesian Network-based Event Correlation

Bayesian networks are one of the most widely used graphical models for representing and reasoning about the probabilistic causal relationships between variables as explained in Kavousi and Akbari [78]. Bayesian networks are usually represented by directed acyclic graphs. Directed acyclic graphs are finite directed graphs that contain no direct cycles. This means that there is no way to start from a given node, and via the directed edges return back to the same node. Each node in the network represents a variable of interest and the edges describe the relations between these variables. The Bayesian network is split up into two parts. First there is the graphical model of the network which shows the nodes and the edges that connect them. Secondly, there is the conditional probability tables associated with each node. The table consist of the probabilities that a node is in a given state given the state of its parent nodes.

Both the research done by Kavousi and Akbari [78] and Qin and Lee [79] utilize Bayesian networks to create "Bayesian attack graphs" (BAG) which are models that use Bayesian networks to depict the security attack scenarios in a system.
As a simple experiment using a Bayesian Network for detection, we have the directed acyclic graph as shown in Figure 2.12. The nodes are a bit like the ones represented in Codebook-based correlation 2.2.5 where the nodes B and C represent two symptom events that are analyzed by the system, these can be events from an IDS, host machine logs, web logs, et cetera. The node A represent a problem node and is not connected to any specific events. The purpose of this Bayesian network, is to answer the following question: What is the probability that, when we observe the two events B and C, we have a problem A?
To calculate this, we first need the the conditional probability tables, which are given in Table 2.4.

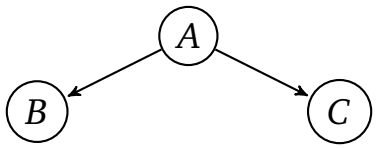


Figure 2.12: Simple example directed acyclic graph

$P(A=0)$	$P(A=1)$
0.8	0.2

A	$P(B=1\|A)$	$P(B=0\|A)$
1	0.9	0.1
0	0.05	0.95

A	$P(C=1\|A)$	$P(C=0\|A)$
1	0.95	0.05
0	0.05	0.95

Table 2.4: Conditional probability tables

We can then calculate the probability that A has occurred, given that we have observed the events B and C by using Bayes' theorem.

$$
\begin{aligned}
&P(A=1|B=1,C=1)\\
&=\frac{P(A=1)P(B=1,C=1|A=1)}{P(B=1,C=1)}\\
&=\frac{P(A=1)P(B=1,A=1)P(C=1,A=1)}{P(A=1)P(B=1|A=1)P(C=1|A=1)+P(A=0)P(B=1|A=0)P(C=1|A=0)}\\
&=\frac{0.2\cdot 0.9\cdot 0.95}{(0.2\cdot 0.9\cdot 0.95)+(0.8\cdot 0.05\cdot 0.05)}\\
&\approx 0.9884
\end{aligned}
$$

In this case, we see that there is a 98.8% chance that the problem/alert A has happened, by observing the arrival of the two events B and C.

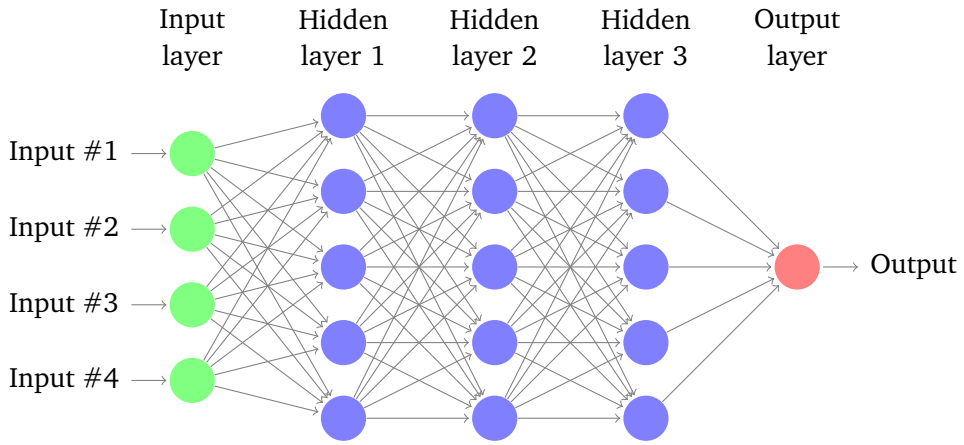


Figure 2.13: Example of neural network with three hidden layers

2.2.8 Neural Network Approaches

Artificial Neural Networks are used in the field of Artificial Intelligence as a system that is inspired by the neural networks in biological brains as explained by Chen *et al.* [80]. These system often come in the form of highly interconnected, neuron-like processing units. As illustrated in Figure 2.13, the circular node represents an artificial neuron and an arrow represents a connection from the output of one artificial neuron to the input of another. These systems are meant to learn and perform tasks by ingesting training data, and creating their own decision model that will be applied when considering future cases.
The computation done in each node can vary from simple mathematical operations like a summation of all its inputs, or by using more complex operations like treshold values, temporal operation as explained by Lippmann [81] or operations that involve the memory of a node as shown in Meira [48]. To allow the network to learn, input weights are often dynamically adapted as stated by Lippmann [81]. Which strategy is used for operation selection and input weighting depends on the application of the network, and multiple approaches exist for this.
As Pouget and Dacier [82] stated in their paper 'Alert correlation: Review of the state of the art', "Neural Networks seem not to be frequently applied in Alert Correlation tools.". The primary reason for this is that it is hard to get insight into how a neural network arrived at the output it produced. Regardless, there are several papers that use artificial intelligence and neural networks for event correlation. The authors of 'Combating advanced persistent threats: From network event correlation to incident detection' Friedberg *et al.* [83] automatically generated a system model with the ability to continuously evolve itself. The proposed approach was able to detect anomalies that are the consequence of realistic APT attacks. In the work by Lin *et al.* [84], the authors used a distributed gradient boosting library to classify real-world malware programs with more than 99% success-rate. Another approach is presented in 'Using neural networks for alarm

correlation in cellular phone networks', where the authors Wietgrefe *et al.* [85] used a neural network to correlate alarms in a cellular phone network.
One of the primary benefits of using neural networks is the ability the networks has to adapt either via training data, or in real time during processing of live event. As pointed out by Pouget and Dacier [82], the main drawback speaks to the fact that it is hard for an analyst to comprehend how a artificial neural network has concluded, which may affect the trust in the system.

2.2.9 Hybrid approaches

In additional to all the "pure" correlation techniques, there also exist various implementations that take a hybrid approach to event correlation by utilizing two or more techniques at the same time. Some examples include the work done by Hanemann and Marcu [86] which combine rule-based event correlation and case-based reasoning, the authors of 'Extracting attack scenarios using intrusion semantics', Saad and Traore [87] proposed a hybrid event correlation approach that used semantic analysis and a intrusion ontology to reconstruct attack scenarios. Furthermore, Ficco *et al.* [88] developed a hybrid, hierarchical event correlation approach for detecting complex attacks in cloud computing. Finally Mé *et al.* [89] to proposed a fully functionalIDS based on event and alert correlations by implementing a language driven signature based correlation that uses FSM to implement the multi-pattern rule matching detection algorithm.

2.3 Simple Event Correlator

As previously stated, throughout the relevant research done with regards to event correlation of system logs, SEC seems to be the most commonly referenced and used software. It is widely used and as Vaarandi [90] explains, has been deployed in several different sectors and industries (Finance, Telecom, IT security, Government, Retail, etc.). SEC has been utilized for several different purposes like fraud detection, insider-threat detection, system fault and availability and security events.
SEC is quite versatile, as it is agnostic to the type of log event that it receives. SEC uses rules that are using Perl-style regular expressions for matching events and extracting data from the event itself using sub-expressions. The extracted data can then be used to correlate between other matching events.
The rules used in SEC are heavily based on regular expressions, which makes it hard to understand, modify and write new rules. The argument for using regular expressions builds on the assumption that most system and network administrators are already familiar with the regular expression language as stated in Vaarandi [91]. Although that might be the case, complex regular expressions can be hard to comprehend, and the output of the regular expression also requires detailed knowledge of what the input event looks like. The rule format of SEC will be further explained under Section 2.4.1. In addition to this, there are few open source

rules and rule-sets with a focus on security, which means that the analyst generally has to start from scratch writing their own rules.
Perhaps the biggest drawback of SEC is that SEC bases its correlation time on when the event was read from the input file. It does not take into account any timestamps that may be in the logs. If logs are ingested from multiple systems (like in a enterprise environment) the logs could be delayed for multiple reasons, or if SEC is unable to ingest the log events fast enough (either because of I/O delays or a huge amount of logs), the timestamp of the logs will be different from when the log event was *actually* produced. The consequences of this could be severe, as events that *should* be correlated together in a given timeframe might drift away from each-other and not be correlated at all.
Scaling is possible, but a bit hard. It is possible to spawn several SEC instances that ingest their own separate event streams and different rule sets as described by Vaarandi *et al.* [92]. Lang [93] utilize this fact to run several instances of SEC on several servers, but also on a single machine as show in Figure 2.14. However this makes it impossible to correlate across event streams, as the SEC instances do not have any knowledge of the other instances in the system. At first Lang [93] considered rewriting and implementing a memory object caching system named "memcached" [94] as seen in Figure 2.15 that would allow the SEC instances to share their context between each-other. However they chose not to tackle that particular problem. In the end, Lang [93] ended up with implementing a solution similar to Figure 2.16, where each SEC instance produces new syslog events and sends them to a master instance which then correlates across the event streams and creates a single alert output.

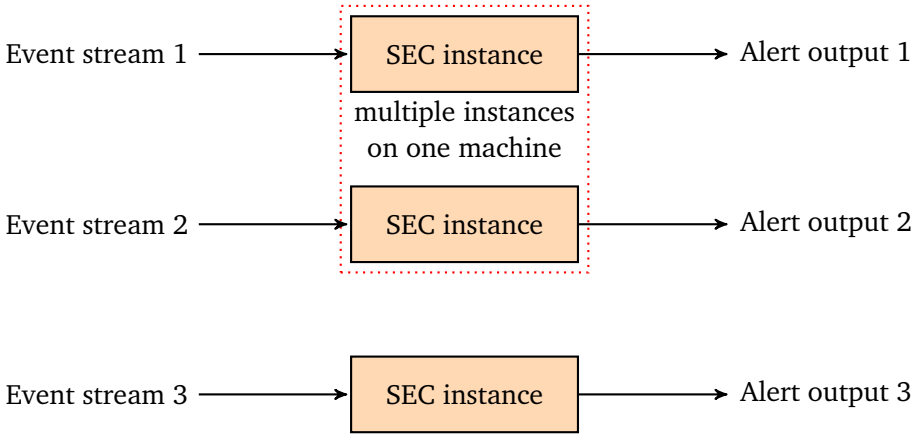


Figure 2.14: Standard SEC usage

2.4 Correlation rules

Just as there are multiple different software and systems for doing rule-based event correlation, there are multiple ways of representing the rules in a knowledge

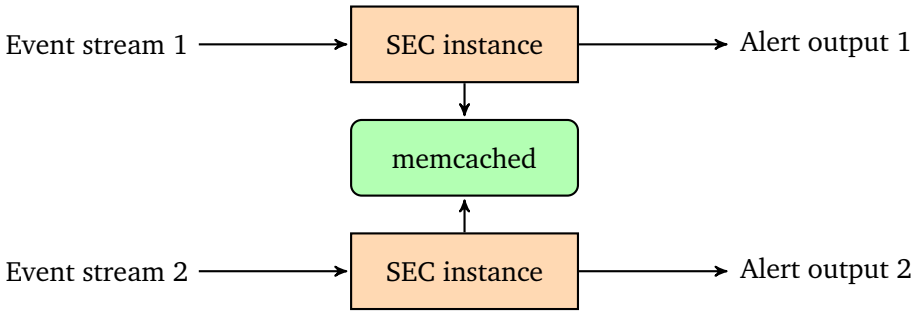


Figure 2.15: Distributed SEC concept

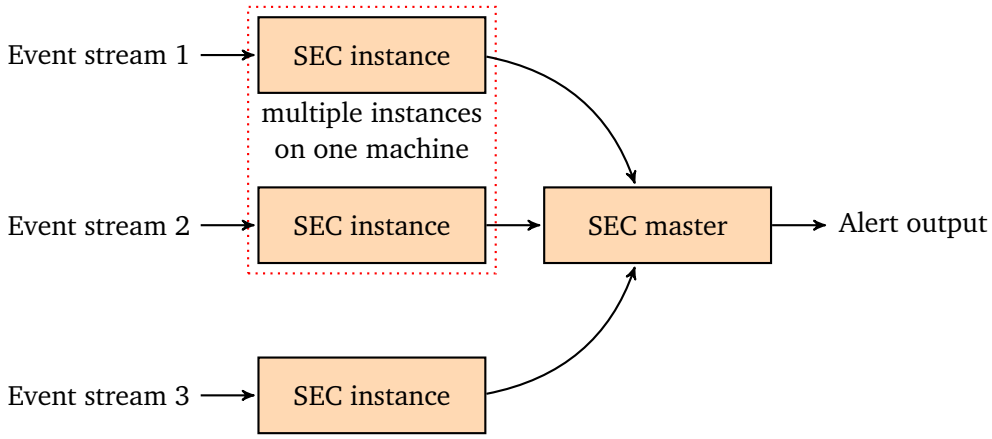


Figure 2.16: Horizontal scaling of SEC

base. In general there are two requirements when it comes to rules. Since they are written and maintained by a knowledge engineer, making the rules readable and easy to create is important. In addition, the rules have to be flexible enough such that it is possible to represent the wishes of the subject-matter expert, preferably without affecting the readability of the rule.
One of the most common ways to write rules are by using boolean operations, either explicitly or implicitly. As exemplified earlier, if we want to write a rule for the occurrence of two different events X and Y, it could be spelled out like "***IF*** *event X* ***AND*** *event Y* ***THEN*** *doAction*". Further complexity could be added by adding additional boolean operations, and by using order of operation marks like parentheses. An exmaple of this could be "***IF*** *event X* ***AND (****event Y* ***OR*** *event Z)* ***THEN*** *doAction*".
Rules for event correlation has been implemented in a range of various ways. General purpose languages such as Lua or Python has been used for example in Prelude [55]. Markup languages like XML and YAML as seen in OSSEC [95], OSSIM [96] and Sigma [97]. Structured Query Language (SQL) rules like those used in Esper [98] or custom definitions like those seen in SEC [54], EQL [99] and Splunk [7].
When the inference engine tries to find matching rules from the knowledge base, the engine might take the linear approach and try each and every rule in sequence until it finds a match (or does not find a match). Another approach to this is by using the Rete algorithm found in Forgy [100]. This algorithm creates a directed acyclic graph that represents the rule set. The graph is defined with a right and left side, called alpha and beta respectively. All the selection and conditional nodes are in the alpha side, while combining and enrichment nodes are on the beta side. As can be seen in Figure 2.17, When a new event is sent through this graph, it enters at the root node in the alpha side of the graph. After passing through the graph, the event ends at a terminal node which is the output of the Rete. As opposed to linear searching the knowledge base, Rete is independent of this and could perform much better when there are a lot of rules involved in the correlation, as told by Pouget and Dacier [82]. Some practical implementations use Rete for event correlation as seen in Doorenbos [101]. The interested reader is referred to Forgy [100] for more details about the Rete algorithm.
As said, Simple Event Correlator (SEC) uses a custom format design particularly for SEC based on regular expressions. We will further examine this rule format under Section 2.4.1. We will also consider Sigma in greater detail in Section 2.4.2 as a possible candidate for replacing the rules used by SEC.

2.4.1 SEC rule format

SEC rule files are simple text-files that contain one or more blocks of key-value pairs. One block is considered one rule. This block contains a set of pre-specified keys that make up how the rule works.
SEC applies each rule sequentially, and will stop looking when it finds a match

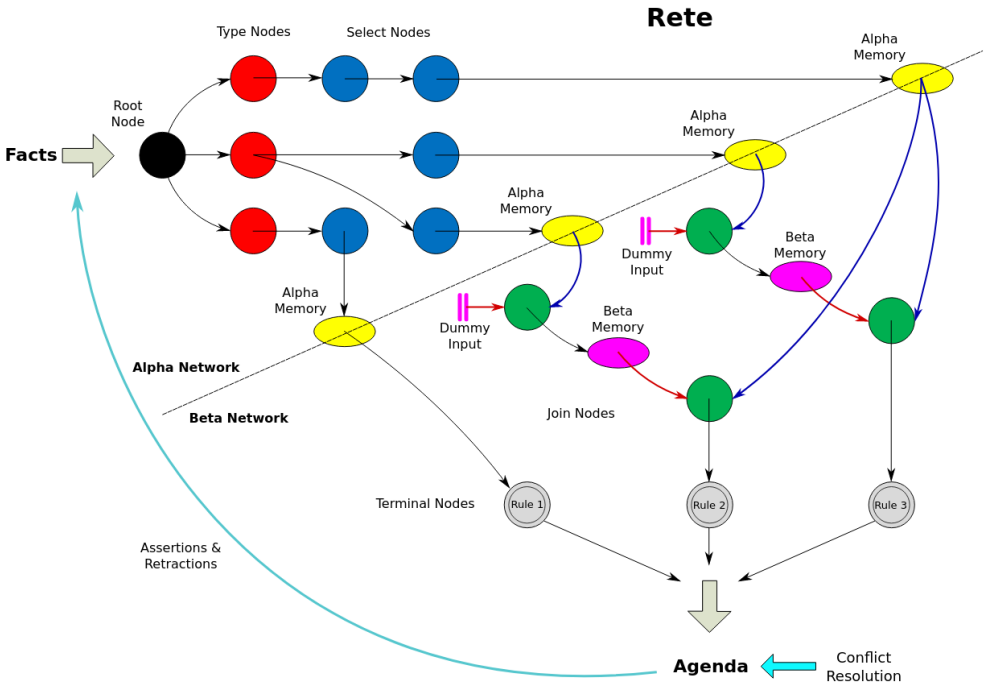


Figure 2.17: Illustrates the basic Rete[102]

(unless that specific rule uses the continue keyword). With this knowledge it is possible to optimize the rule set by placing more popular rules nearer the top of the rule set as told by Rouillard [103].
A rule consists of a subset of the following keywords, where some of the keywords are only applicable if some of the keywords holds a special value:

- type - which kind of rule
- ptype - which type of pattern
- pattern - pattern to match event against
- desc - rule description
- action - action to take if pattern is matched
- continue - if set, allows SEC to continue searching for other matching rules
- context - boolean statement based on global context variables
- thresh - (if applicable) threshold for triggering event
- window - (if applicable) time window in seconds

for **type**, there are multiple different possible values:

- Single - Match input and execute action.
- Suppress - Suppress the matching input which keeps the input from being matched by later rules.
- Calendar - Execute action on a given time.
- SingleWithSuppress - A combination of Single and Suppress. Match input and execute action, but suppress the matching input for a set period of time

afterwards.
- Pair - Match input and execute action, then wait until another event arrives and execute second action.
- PairWithWindow - Like Pair, but also execute an action if second event does not arrive.
- SingleWithThreshold - Count matching input in a time window, if number of matches is above a threshold, execute action and ignore matches for rest of window.
- SingleWith2Thresholds - Count matching input in a time window, if number of matches is above a threshold, execute action. Then create new count, and if number of matches drops below the threshold, execute action.
- EventGroup - Count N number of different events and execute action if all of them reach their given threshold.
- SingleWithScript - Match value and depending on return-value of external script, do action.
- Jump - Submits matching event to another rule set for further processing.

For **ptype**, there are multiple different possible values:

- SubStr - pattern is a string that will be searched for in the event
- RegExp - pattern is a Perl regular expression
- PerlFunc - pattern is a Perl function for matching
- Cached - pattern matches previously cached patterns.
- TValue - pattern is a boolean value (TRUE/FALSE) that always or never matches.

In addition to the above-mentioned values for ptype, they all have (except for TValue) a negated version as well, prefixed with "N" (as in NSubStr, NRegExp, etc).
The following example in figure Code listing 2.1 is a slimmed down version of the "MITRE CAR-2013-04-002: Quick execution of a series of suspicious commands" [104]. The purpose of the rule is to detect quick execution of commands that a regular user would not frequently do, but that an attacker might run as part of their reconnaissance or exploitation of the system.
As an example, consider that the following events occur within 10 seconds from start to finish:

1. Alice ran word.exe on PC1
2. Bob ran calc.exe on PC2
3. Mallory ran whoami.exe on PC1
4. Mallory ran ssh.exe on PC1
5. Bob ran powershell.exe on PC2
6. Alice ran firefox.exe on PC1
7. Mallory ran powershell.exe on PC1

8. Mallory ran systeminfo.exe on PC1
9. Bob ran word.exe on PC2
10. Alice ran powerpoint.exe on PC1
11. Mallory ran hostname.exe on PC1

If we apply the rule shown in Code listing 2.1, during the execution, the following events will be created and re-injected into the event stream:

1. Interesting_commands_by_Mallory_on_PC1
2. Interesting_commands_by_Bob_on_PC2
3. Interesting_commands_by_Mallory_on_PC1
4. Interesting_commands_by_Mallory_on_PC1

These re-injected events will be processed by rule #4, and when the threshold number of 3 is met for the event "*Interesting_commands_by_Mallory_on_PC1*", the rule will trigger its action and write "*Three interesting commands were run on host PC1 by user Mallory*" to the console.

Code listing 2.1: Example ruleset for detecting quick execution of a series of commands

```
# Rule 1
type=Single
ptype=RegExp
pattern=(\S+) ran whoami\.exe on (\S+)
desc=$0
action=event Interesting_commands_by_$1_on_$2

# Rule 2
type=Single
ptype=RegExp
pattern=(\S+) ran powershell\.exe on (\S+)
desc=$0
action=event Interesting_commands_by_$1_on_$2

# Rule 3
type=Single
ptype=RegExp
pattern=(\S+) ran hostname\.exe on (\S+)
desc=$0
action=event Interesting_commands_by_$1_on_$2

# Rule 4
type=SingleWithThreshold
ptype=RegExp
pattern=Interesting_commands_by_(\S+)_on_(\S+)
desc=$0
action=write - Three interesting commands were run on host $2 by user $1
window=10
thresh=3
```

In addition to the rule show in Code listing 2.1, we can implement the same rule by using the EventGroup type as show in Code listing 2.2. This rule works similarly

as the first rule, but the main difference with EventGroups is that all the event conditions have to be satisfied before the action will trigger. This means that we explicitly have to have all three patterns match, and will not trigger if for instance *whoami.exe* is ran three times in a row.

Code listing 2.2: Example ruleset 2 for detecting quick execution of a series of commands

```
type=EventGroup3
ptype=RegExp
pattern=(\S+) ran whoami\.exe on (\S+)
ptype2=RegExp
pattern2=(\S+) ran powershell\.exe on (\S+)
ptype3=RegExp
pattern3=(\S+) ran hostname\.exe on (\S+)
desc=Three interesting commands were run on host $2 by user $1
actiom=write - Three interesting commands were run on host $2 by user $1
window=10
```

This is not an extensive listing of the features in the SEC rule language, but covers what is needed for the rest of the thesis. For a deeper dive into SEC rules with more examples, the reader is referred to the paper 'Real-time Log File Analysis Using the Simple Event Correlator (SEC).' by Rouillard [103] and the SEC man-pages [105].

2.4.2 Sigma

Sigma is a "Generic Signature Format for SIEM Systems"[97]. Sigma is an open standard for rules that are used to generically describe searches in log data. The value proposition of Sigma is that there is a lack of standardisation within the SIEM search field. A given query to search for the same item might look very different depending on the SIEM platform used. This makes it inherently harder to share and contribute rules within the community.
Sigma is primarily used as a high-level rule that is transcompile into SIEM queries for products like Splunk [7], ElasticSearch [106], NetWitness [9], etc. The rules are written in YAML Ain't Markup Language (YAML)[107], which is a key-value based format that uses indentation to indicate nesting. The rule format contains some required and some optional fields, and it is extensible with custom fields as shown in Figure 2.18.

Code listing 2.3: Example Sigma rule for detecting quick execution of a series of commands

```
title: Quick Execution of a Series of Suspicious Commands
logsource:
    product: windows
    service: sysmon
detection:
    selection:
        OriginalFileName|contains:
            - whoami
            - hostname
            - powershell
```

```
    timeframe: 10s
    condition: selection | count(User) by MachineName >= 3
```

The Code listing 2.3 is a minimal example that is similar to the rule shown in Code listing 2.1. An example input event can be found in Code listing 2.4. The rule is selecting the `OriginalFileName`-key from our event and checking if it contains any of the following entries: `whoami`, `hostname` or `powershell`. The rule creates a timeframe of 10 seconds, and the condition counts the distinct user names grouped by `MachineName`, and checks if the count is more than or equal to three.

Code listing 2.4: Example event for Sigma

```
MachineName: Client01.mrtn.lab
UtcTime: 2020-02-18 10:29:49.839
ProcessId: 1040
OriginalFileName: whoami.exe
User: MRTNLAB\mrtn
```

There exists a vast amount of example rules, and new rules are added continuously to the project by contributors [108]. For further information about the details of the Sigma specification, the interested reader is referred to the Sigma Specification [109].

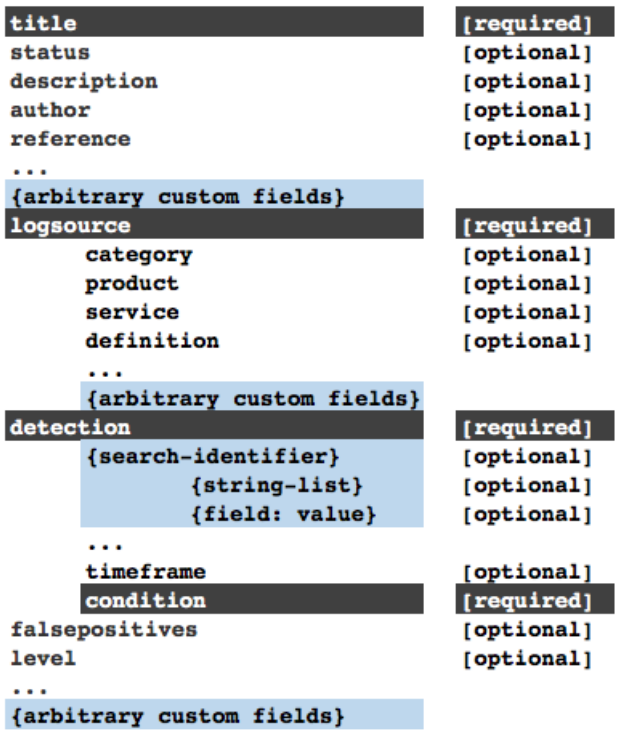


Figure 2.18: Sigma specification [109]

Chapter 3

Methodology

One of the main goals of this thesis is to explore if there is any way that we can improve the way real time event correlation is done and how our improvement compare to other methods. As outlined in Chapter 2 we have chosen to compare our solution against Simple Event Correlator (SEC). In addition to the focus towards SEC, we will particularly look at event correlation of Windows Event Logs. In the following chapter we will discuss the methodology used address these goals. We will evaluate which datasets exist and should be used, we will discuss the various ways we can improve how event correlation can be done, and we will take a look at how that performance change can be measured.

3.1 Datasets

To properly address the research questions proposed, it is important to have one or more datasets that can be used to evaluate the performance of the proposed solution in this thesis. There is not a vast variety of available datasets that focus on Windows Event Logs publicly available, but there are some that have surfaced in the recent years. We will present those in the following section and offers a short evaluation in the context of this thesis.

3.1.1 Evaluation of existing datasets

When evaluating which datasets we want to use for our experiments, we first have to define some parameters that we can measure the datasets by:

- Size - The dataset must be large enough to measure the performance of existing solutions and our proposed solution.
- Representative - The dataset must be representative of the real world
- Up to date - The dataset should preferably be of a recent date

Boss of the SOC

Boss of the SOC (BOTS) are datasets created for Splunks Boss of the SOC capture the flag competitions [110]. The datasets are created in a controlled environment, where some adversarial actions has taken place. The contestants have to analyze and hunt in the data to answer several security-related questions which grant points.
There are currently three different datasets available. Each with a different focus. The first dataset consists primarily of Suricata [50] and Windows events. The second dataset also contains Suricata and Windows events, in addition to more application specific logs like Symantec Endpoint Protection, Weblogic, MySQL etc. The third and last dataset focuses more on cloud and hybrid environments and do not contain the same amount of Windows event logs for instances.
The datasets from BOTS are released in a Splunk pre-indexed format, meaning that one would have to set up a Splunk instance, import the indexed datasets, and then export the datasets out in a more usable format (like JavaScript Object Notation (JSON)).

Mordor

The Mordor datasets [111] are pre-recorded events generated by simulating adversarial techniques in a test environment using common red team tools like Empire and Cobalt Strike.
There currently exists two datasets under the Mordor project, namely APT29 and APT3. These datasets contain Windows event logs from simulated Advanced Persistent Threat (APT) actions. These actions are predefined by the MITRE ATT&CK Evaluations project [112]. The MITRE ATT&CK Evaluations project is created as a way to evaluate different endpoint solutions ability to detect various adversarial techniques, tactics and procedures. The adversarial actions are maps to techniques under 10 categories in the MITREs ATT&CK Matrix [113], as shown in Table 3.1.

Categories
Initial Access
Execution
Persistence
Privilege Escalation Defense Evasion
Credential Access Discovery
Lateral Movement
Collection
Command and Control
Exfiltration
Impact

Table 3.1: List of MITRE ATT&CK Matrix categories

We will focus on the APT3 dataset. This dataset consists of two subsets, one for

each scenario as outlined by MITRE in their Attack Emulation Overview [114] for APT3.

Synthetic datasets

Synthetic data is datasets that are generated and design with the intent to measure some specific condition or event that may not be found in real world data, or that the real world data would be hard to come by, as told by Barse *et al.* [115]. There are multiple reasons why one might consider to use a synthetic dataset, like simulating a large period of time which would be unrealistic to capture in real life, simulating extraordinary events occurring, huge data loads, and so forth. Continuing this section, we will consider three different synthetic datasets that we will be applying during our experiments in Chapter 4.
It is worth stressing that the synthetic datasets are used strictly for measuring the performance of the systems in a worst-case/best-case scenario, and the dataset is in itself not representative of a real world scenario.
Baseline dataset
This dataset is a dataset with events that are all benign. This dataset is useful for measuring the speed at which the tools process and analyse the events, without triggering any rules.
High signal, low noise dataset
If we want to test the maximum event correlation throughput possible, we want to use a dataset that is designed to continuously trigger one or more rules. Given that we want to trigger a rule like the one defined in Code listing 2.3, a high signal, low noise dataset could be designed to repeat the same 3 log lines that are enough to trigger the rule.
Low signal, high noise dataset
This is the opposite of the high signal, low noise dataset which contained the necessary log lines to repeatedly trigger a specific rule. This dataset only contains the necessary events to trigger a single rule once, the rest of the events are simply background noise. This is pretty similar to the baseline dataset.

3.1.2 Datasets used in this thesis

For the experiments conducted in this thesis, Multiple datasets have been chosen:

- Mordor dataset (APT3, Scenario 1 and 2)
- High signal, low noise dataset
- Baseline dataset

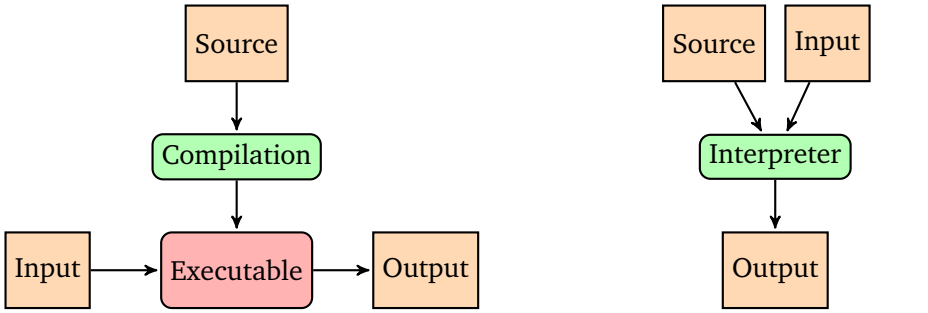
We chose the Mordor dataset as it is sufficiently large enough, representative and up-to-date. Then we have chosen the baseline and high signal, low noise synthetic datasets as they will be used for baselining and giving us best and worst-case scenarios for performance measuring. We do not use the low signal, high noise dataset, as we consider that almost identical to the baseline dataset.

3.2 Improving real time event correlation for Windows Event Logs

With regards to Research Question 2 in Section Section 1.3, the question we are trying to answer is if there are ways we can improve how real time event correlation is done. We will discuss multiple approaches to how this can be achieved in the following section.

3.2.1 Compiled language vs. interpreted language

As previously stated in Section 2.3, SEC is written in the Perl language. Perl is an interpreted language that according to its creator Wall *et al.* [116] is "optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information". Being an interpreted language means that the code is not compiled into machine code and executed like a compiled language does, but the interpreter parses the code step-by-step and execute its actions in subroutines. We can see an overview of both in Figure 3.1.
There are many benefits to choosing a interpreted language. The interpreter can hide a lot of the complexity when programming, which means that a interpreted language can be easier to write, use and understand. Similarly to the benefits seen with the rules in rule-based event correlation Section 2.2.2, the programming language can be written with a close similarity to the human language. Additionally, the programs can run cross-platform, as the interpreter manages the lower level details of the specific architecture that we are executing code on.
The main disadvantage is the additional overhead required by the interpreter. Compiled code will generally always be faster than interpreted code, because it runs closer to the "bare metal". When we want to increase performance, working with compiled languages are generally considered the right thing to do.
In the compiled language world, C and C++ has been the kings for a long time. In recent years, other languages like Go [117] and Rust [118] has seen the light of day, and are increasing in popularity. Benefits of the new generation of compiled languages is the built in features for memory safety, safe concurrency, security and better designed languages that makes it easier to get started with the language. This has been the main issue with traditional compiled programs, they are harder to write and easier to get wrong than a program written in a interpreted language.
While this section might give the impression that there is a black and white difference between compiled and interpreted languages, that is not technically correct. In modern times, languages such as Lisp and Pascal implement both, and Java and C# are compiled into an intermediate language (bytecode) which is executed in a virtual machine as described in Henriques and Bernardino [119]



(a) Compiled language model **(b)** Interpreted language model

Figure 3.1: Illustration of compiled vs. interpreted language

3.2.2 Concurrent execution

As discussed, SEC is not taking full advantage of the system when only running in a single thread. It is a fair claim that by using multi-threading it is possible to increase the throughput of an alternate solution which will process events much quicker. We can symbolize the difference with the synchronous example in Figure 3.2, and the concurrent process as seen in Section 3.2.2.
While they both process the same amount of events, the concurrent version handles the total number of events much quicker than the synchronous version. Note that it is not given that each individual event is processed any faster in any of the two examples. In fact, given that we probably want to correlate between the events, the concurrent version could use longer time to handle each event, as it has to check a shared context between the threads, which could cause some overhead.

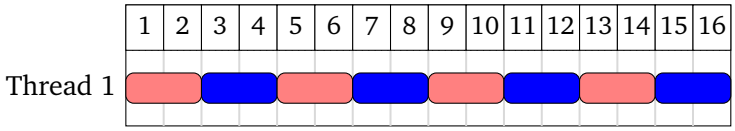


Figure 3.2: Synchronously processing of 8 events

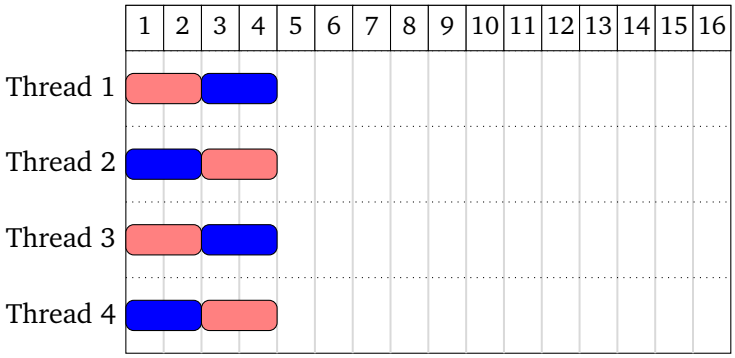


Figure 3.3: Concurrent processing of 8 events

3.2.3 Better rules

The rules in the knowledge base is the bread and butter of the rule-based event correlation. And although there are multiple different ways to create rules as discussed in Section 2.4, it is always worth considering if other rule formats could be more beneficial. As stated by Rouillard [103], the majority of the computational time used in SEC is spent on matching events against regular expression, so if we could in some way remove the need for the extensive use of regular expressions by using another rule format, that could potentially be a much faster solution.
We outlined a possible rule candidate in Section 2.4.2, namely Sigma. We will look at implementing Sigma when we experiment with the rule change.

3.2.4 Proper time management

One of the biggest drawbacks of SEC as outlined in Section 2.3, is the fact that when it bases its correlation time on when the event was read from the input file. It does not take into account any timestamps that may be in the logs. If logs are ingested from multiple systems (like in a enterprise environment) the logs could be delayed for multiple reasons, or if SEC is unable to ingest the log events fast enough (either because of I/O delays or a huge amount of logs), the timestamp of the logs will be different from when the log event was *actually* produced. The consequences of this could be severe, as events that *should* be correlated together in a given timeframe might drift away from each-other and not be correlated at all.
Instead of the time being based on when the event is read, we want to base our correlation on when the event was generated by the source system. Since we are doing the assumption that we will only be working with Windows event logs which have the UTC timestamp in the logs, we are able to use that. However, if we were to expand to ingest other logs as well, we would have to take into account that the time might be represented differently in the log. It is rare to see logs that do not have a timestamp in some form or fashion. The hardest part might be localization, if the timestamp is not written with a specific timezone. However, this will not be a problem in this thesis as all Windows event logs are written with the UTC timezone.

3.2.5 Internal representation of logs

When we are testing the different rules in SEC against a log line, the pattern of the rule is applied against the whole log line. We propose that tokenizing the log before testing each rule could improve the performance.
Tokenizing the log means that we are taking a log in the form of "EventID: 1
nMachineName: client1
nUser: john", and parsing it into a object instead, as seen in Code listing 3.1. The benefit of this is that we can query specific parts of the event log directly, instead of having to parse the whole event log every time we want to access a

single key-value pair. An example could be if we wanted to access the MachineName or User values from Code listing 3.1, which could do something like this `event['MachineName']` and `event['User']`.

Code listing 3.1: Example tokenization

```
Object event = {
    EventID = 1
    MachineName = client1
    User = john
}
```

Moving away from the large regexes as already discussed in Section 3.2.3 and using tokenization to enable using new rule formats like Sigma could improve the performance of our solution.

3.2.6 Support for multiple log formats

As briefly discussed in Section 3.2.4 the biggest hurdle would be event logs that either do not contain a timestamp, or are syntactically hard to parse or tokenize as discussed in Section 3.2.5. In this thesis we are focused on Windows Event Logs, but it is possible that other log sources would be possible to have working without any or little change to the solution this thesis proposes. We consider this future work.

3.2.7 Output modularity

Defining different alert output channels. It would be nice to be able to create granular output rules that takes some decision based on the alert severity and sends the alert to the proper channel. Channels could include:

- E-Mail
- Instant Messaging platforms like Slack, Teams et cetera.
- Ticketing system
- SIEM products like Splunk

We have chosen not to implement these as we focused primarily on performance measurements, and consider this future work.

3.2.8 Distributed correlation

There are multiple reasons why we might consider using a distributed correlation system. A distributed system first of all provides redundancy if one or more ingestion node or correlation server should fail, having the system continue running without experiencing loss in data. This is important because when we are correlating, we never know when a rule might hit, and any loss of data or interruptions in the correlation process can lead to missed alerts. Furthermore, with regard to geolocation, being able to reduce latency by ingesting data from hosts as close to them as possible could improve the real time effectiveness of the system. Any

system should be able to handle delayed data, but having as little delay when ingesting data is still preferable. Lastly, if we want to scale up our system to handle bigger loads of data and correlation rules, we need scalability.
As discussed in Section 2.3, the authors of Lang [93] considered a few ways to scale SEC as shown in Figure 2.15 and Figure 2.16.
When scaling a system, we generally consider two different types of scaling. Horizontal and vertical. Horizontal scaling means that we are adding additional machines into a pool of resources for that particular service. Vertical scaling is adding more power to the existing machine, for instance by increasing the available RAM or upgrading the CPU(s). There are multiple considerations that have to be done when choosing which way to scale a system. Horizontal scaling usually comes with the drawback of having to manage the pool of resources for each scaled service. Vertical scaling is in general simpler, but at some point it is no longer possible or feasible to scale further with regards to performance and cost. The implementation show in this thesis is primarily built to scale vertically. Interesting future work would be to add horizontal scaling to the proposed solution in this thesis, much like in Figure 2.15, and tackle the challenges associated with load balancing, shared "context memory" between the correlators, and other possible obstacles.

3.3 Measuring performance

There are multiple factors that affect the performance of event correlators. All these factors lead to multiple ways that we can measure performance. This section tries to outline the most important ones.

3.3.1 Data ingestion speed

At the start of the data pipeline, we have to ingest our data for processing. Data ingestion is the process of importing data from an external source into our program. The rate at which we are ingesting events are usually measured in events per second. Data ingestion is based on a emitter sending the data, and a receiver receiving the data. The emitter does not have to be a separate system, it can be a hard disk, RAM or a network-based service. The performance related to data ingestion speeds can be bottlenecked by several possible things. The emitter may be bound by the storage medium it is sending data from. If we are reading events from a log file stored to disk, we are bound to the read speed that our disk(s) support. If we are reading events from a process that stores the events in memory, we are bound by the read speeds of our RAM. With regards to network-based transmission, the choice of transport-layer protocol used can have an impact on transmission speed. Using UDP may be the fastest, but could lead to dropped packages which not optimal. Using TCP ensures that all events are transmitted, but at the cost of additional overhead for re-transmitting lost data, re-arranging packets out of order, et cetera. When transmitting data in a network (both internal and via the internet) encryption is needed to ensure authenticity and tamper protec-

tion of the data. But encryption comes at a cost, namely that it takes some time to encrypt and decrypt data when its sent and received. Additionally, the networking hardware can play a role depending on the setup. The supported speeds of the network card in the emitter and receiver, and any intermediate networking equipment like switches and routers could affect the throughput of events. There are multiple ways of transmitting data over the network that may affect the ingestion speed, and that is the implementation of how transmitting shall be done. Real-time transmission sends the events as soon as they happen, one-by-one. Another tactic is to use batching or chunking that sends bursts of events instead of sending the events one-by-one. Finally, a hybrid approach is possible where the emitter chooses which type to use depending how many events are being transmitted. Then we have the ingestion capabilities of the receiver. This boils down to how efficient the receiver is to manage the backlog of events it receives. A simple program might only allow processing one event at the time, blocking incoming new events. A more efficient program might store a backlog of events in RAM, which ensures that it does not block incoming new events.
There are multiple ways we can measure all these different possible bottlenecks. For disk and RAM-based operations we can use profiling tools that come with the operating system to measure the load we are under. We can look at the number and size of the network packages being sent and received. Given an external emitter running for instance the software Kafka, we can get an overview of how fast receivers are fetching data from the emitter. Likewise, we can do the same from the end of the receiver by looking at how many events we are ingesting into our program per second. Finally, we can test the ingestion using timing, by ingesting a set number of events and timing how fast the receiver can ingest them (without any processing other than ingesting), we can calculate the number of events per second.

3.3.2 Processing speed

Since the ingestion speed discussed in Section 3.3.1 might fluctuate depending on how the data is ingested, measuring the internal processing speed might be more interesting when evaluating the performance of the various solutions. This removes the uncertainties related to ingestion speeds. There are multiple options when looking at internal processing speed. One might look at the processing as a whole from start to finish, or try to separate out the various internal steps that occur during processing. Go features a profiler that will output a graph, showing which functions are taking up the most time during runtime. This can give an idea of where the most of the time is spent during processing.
The processing speed can be affected by several things. First of all the dataset used will matter, as the number of matches will have an effect on the number of alerts generated and contexts updated. Secondly, the implementation of how rules are processed and checked against events can have a big impact on the processing speed. If the solution is able to quickly disregard events as not interesting, there is

a big potential for saving time. Lastly, the internal handling of contextual information, how that information is accessed and other performance-related improvements all have an effect on the processing speed observed. The biggest drawback of this approach is the need profiling or timing "inside" the solution. While this might be simple to implement in a new solution, patching such a feature into older solutions can prove to be hard or in the worst case error-prone if the solution being patched is not fully understood.

3.3.3 Compound processing speed

Measuring the compound processing speed will give us a bird's-eye view of what the total processing speed is. It takes into account both ingestion and processing speeds, measures the total time used, including both I/O and any internal processing.
Depending on the solutions, this might be the best or only option for a good one-to-one comparison between solutions, if they do not support the same ingestion abilities.

3.4 Test plan

As discussed in this chapter we have identified multiple ways that possibly could improve or further expand the capabilities of existing real time event correlation solutions, more specifically SEC.
In Chapter 4 we will be using the Mordor [111] APT3 dataset, in addition to three synthetic datasets as explained in Section 3.1.
We will focus our experiments around the possible improvements mentioned in this chapter, namely using a compiled language, utilize concurrent execution, test if better internal representation of log data and using other rules might affect performance and lastly implement better time management. Distributed correlation, output modularity and support for multiple log formats is considered future work.
As discussed in Section 3.3, there are multiple ways that we can measure the performance of our solution against existing solutions. We will be using compound processing speed as discussed in Section 3.3.3 for our performance tests.

Chapter 4

Experiments

The following chapter introduce our improved implementation based on the methodology presented in Chapter 3. The software and hardware specifications are listed, the dataset collection and required preprocessing is presented, and we introduce our solution in two step, first a solution that uses the same rule format as SEC, and then a improved version that implements Sigma [97] and a better way for internally representing events as discussed in Section 3.2.5.

4.1 Hardware and Software Specifications

The host system used for running the experiments feature a Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz processor and 24 GB memory. The processor features two physical cores, and is capable of running two threads per core. This means that the processor has a maximum of 4 logical cores.
The software versions of interest are:

- Ubuntu 18.04.4 LTS, released February 2020
- go version go1.13.3 linux/amd64, released October 2019
- Perl v5.30.2 built for x86_64-linux, released March 2020
- Simple Event Correlator v2.8.2, released on Jun 2, 2019

4.2 Dataset preprocessing and analysis

In total, the two subsets contain 223 563 log lines in JSON format. 116 572 of these are of the type "Microsoft-Windows-Sysmon" which will be the main focus of our experiments. As previously explained in Section 2.3, SEC is created to work with logs that contain one event per line in syslog format. For us to be able to use the Mordor dataset in SEC, we had to convert the JSON logs into a syslog-friendly format. We converted the Mordor APT3 datasets by extracting the hostname and the raw Windows Event message which was still intact in the JSON events. The script used can be found in Appendix A.
It was interesting to us to graph the dataset, as a way to identify if the frequency

of events are relatively stable, or of there are peaks in the dataset. Using the script found in Appendix B we calculated how many events occurred in every 10 second interval in the dataset. This is valuable as it will tell us what the peak number of events might be, and will guide us in understanding if we are reaching our goal of real time event correlation. We chose 10 seconds because our example rule (as seen in Code listing 2.1) uses this number as its time window. In addition, we wanted to look at the number of computers and users in the dataset. This is valuable as it will give us an idea of how large the environment is. We did this using the scripts in Appendix D and Appendix C respectively.
Finally, it might be interesting for us to see how our implementation handles multiple rules, and if that impacts performance. The script used for generating 1000 events can be seen in Appendix G.

4.3 Implementation that uses SECs own regex-based rule format

4.3.1 Choosing a compiled language

As explained in Section 3.2.1, there are several benefits when using a compiled language in terms of performance gains. We landed on Go as our language for implementing our new solution.
Go [117] is cross-platform, supports garbage collection, strongly and statically typed. In addition, Go features powerful built-in profiling tools and race-condition detection that can help development. This is especially valuable as we know we want to implement concurrency, and detecting and fixing any race-condition issues is of great importance. Go makes building concurrent programs easy by providing features such as goroutines for spawning new threads, and channels for communicating between the threads. This will not be an extensive intro to Go, the interested reader is referred to Go [117] for further details.

Goroutines, channels and workers

Goroutines are not "real" threads. They are lightweight threads managed by the Go runtime, with a lower cost of creation than regular threads [120]. Channels are the preferred way to communicate between goroutines in Go, and are created to prevent any race conditions when multiple goroutines are reading and writing to the same channel. The use of channels and goroutines gives us the ability to run safely in a threaded matter, utilizing multiple cores. Since goroutines run in the same address space, any access to shared memory outside of channels has to be synchronized to avoid race conditions or data races.
Continuing forward in this thesis, we will use the term *worker* for a goroutine that is created to handle events. By spawning multiple workers, we are able to handle a bigger workload and increase the event throughput of our implementation.

4.3.2 Implementation

When considering which features we wanted to implement from SEC, we chose to implement the features that we saw the best value in. We chose to only implement the *Single* and *SingleWithTreshold* type, and the *RegExp* pattern type. These are the features required to implement the rule found in Code listing 2.1, and also some of the most popular features observed from the SEC rule repository [121]. For testing this implementation, we used the rule found in Appendix E.
Furthermore, we implemented threading by using goroutines and channels. The architecture can be seen from the Figure 4.1. While it might seem complex, in reality it is pretty simple. Each block is a separate Go routine running in a light-weight thread. `getEvents()` reads events from input, and sends each event on a channel named `eventChannel`. The `handleEvent()` goroutines (named workers in our implementation), listens to this channel and when a new event arrives, picks it off the channel and starts processing it. As can be seen from Figure 4.1, the workers are sharing context, that they will lock on if any rules are matching and they need to do some correlation. If a rule matches and issues a *event* action (as shown in Code listing 2.1), the worker will push the event action on to a new channel that is being listened to by `reinjectEvents()`. `reinjectEvents()` is a Go routine with the sole purpose to collect events from multiple workers and forward them on a single channel, reinjecting into `eventChannel`. This makes the new events available to the the workers, so that they can process the new events. If any of the `handleEvent()` workers completes a correlation according to the rule, and the rule issues a *write* action, the action is written to output.
When we want to do correlation between two or more events based on a rule, we need to have some kind of overview of what state our rule is in. In Figure 4.1 we denote this as *context*. When a new event arrives that triggers our rule, we need to know if this is the first event, if there are other events that have triggered before it, and most importantly, if the previous events that triggered the rule is within the given time frame of the rule.
One of the benefits of our new implementation is the ability to process events concurrently. But when working with a context that is accessed by several workers concurrently, data races may appear. A data race occurs when two goroutines concurrently accesses the same variable (in this case the context variable), and at least one of the goroutines writes to the variable. The danger here is that we could have two or more goroutines with their own versions of the context that are out of sync. This could lead to data loss and/or a failure to detect when a rule-condition is met. The standard way of dealing with data races like this is to use a mutex. A mutex provides a locking mechanism to ensure that only one goroutine can manipulate a variable at a time.
In our implementation we integrated a per-rule mutex. This gave us a goroutine-safe way of accessing and editing our context. It is safe to use this as a lock, since a worker only will we working with one rule context at a time. If several goroutines are accessing the context at the same time, but are interested in different rules,

we will lock on the individual rule mutex instead of having to lock on a single shared mutex which would lead to more waiting for locks to unlock.

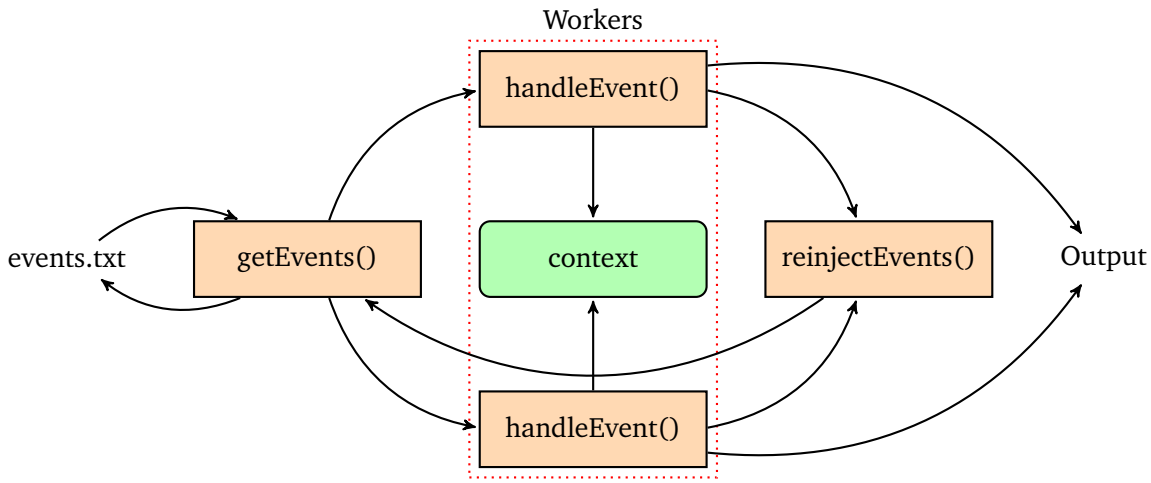


Figure 4.1: Reimplementation in Go

4.4 Implemented a new rule format

As stated, we wanted to create another version that implements Sigma [97] and a better way for internally representing events as discussed in Section 3.2.5.
As discussed in Section 3.2.5, SEC and our implementation in Section 4.3, when tested the different rules against a log line, the pattern of the rule is applied against the whole log line. In Section 3.2.5 we proposed that tokenizing the log before testing each rule could improve the performance. When we tokenize the event log, we take a single line of log/event, and split it into its key-value representation. For instance, the event log found in Code listing 4.1 is a huge single line of text. Both writing rules for, and using regular expressions, on such a large log line seems inefficient.

Code listing 4.1: Example syslog event

```
<14>Feb 18 02:29:49 Client02.mrtn.lab Microsoft-Windows-Sysmon[2092]: Process
    Create:  RuleName:   UtcTime: 2020-02-18 10:29:49.839  ProcessGuid: {dadb16ad-
    bc9d-5e4b-0000-0010c8fd3600}  ProcessId: 1040  Image: C:\Windows\System32\
    whoami.exe  FileVersion: 10.0.17763.1 (WinBuild.160101.0800)  Description:
    whoami - displays logged on user information  Product: Microsoft Windows
    Operating System  Company: Microsoft Corporation  OriginalFileName: whoami.exe
     CommandLine: whoami  CurrentDirectory: C:\Users\mrtn\  User: MRTNLAB\mrtn
    LogonGuid: {dadb16ad-2c2d-5e17-0000-0020fc3c1b00}  LogonId: 0x1B3CFC
    TerminalSessionId: 1  IntegrityLevel: Medium  Hashes: MD5=43
    C2D3293AD939241DF61B3630A9D3B6,SHA256=1
    D5491E3C468EE4B4EF6EDFF4BBC7D06EE83180F6F0B1576763EA2EFE049493A,IMPHASH=7
    FF0758B766F747CE57DFAC70743FB88  ParentProcessGuid: {dadb16ad-2cf1-5e17
    -0000-001027122b00}  ParentProcessId: 2748  ParentImage: C:\Users\mrtn\test.exe
      ParentCommandLine: .\test.exe
```

If we tokenize the event before processing, we turn the event found in Code listing 4.1 into something like what we have in Code listing 4.2.

Code listing 4.2: Example tokenized event

```
MachineName: Client02.mrtn.lab
ProcessType: Process Create:
RuleName:
UtcTime: 2020-02-18 10:29:49.839
ProcessGuid: {dadb16ad-bc9d-5e4b-0000-0010c8fd3600}
ProcessId: 1040
Image: C:\Windows\System32\whoami.exe
FileVersion: 10.0.17763.1 (WinBuild.160101.0800)
Description: whoami - displays logged on user information  Product: Microsoft
    Windows Operating System
Company: Microsoft Corporation
OriginalFileName: whoami.exe
CommandLine: whoami
CurrentDirectory: C:\Users\mrtn\
User: MRTNLAB\mrtn
LogonGuid: {dadb16ad-2c2d-5e17-0000-0020fc3c1b00}
LogonId: 0x1B3CFC
TerminalSessionId: 1
IntegrityLevel: Medium
Hashes: MD5=43C2D3293AD939241DF61B3630A9D3B6,SHA256=1
    D5491E3C468EE4B4EF6EDFF4BBC7D06EE83180F6F0B1576763EA2EFE049493A,IMPHASH=7
    FF0758B766F747CE57DFAC70743FB88
ParentProcessGuid: {dadb16ad-2cf1-5e17-0000-001027122b00}
ParentProcessId: 2748
ParentImage: C:\Users\mrtn\test.exe
ParentCommandLine: .\test.exe
```

The tokenized version of the event log is stored as a struct, which makes it simpler to query specific parts of the event log directly, instead of having to parse the whole event log every time we want to access a single key-value pair. An example would be if we wanted to access the MachineName or CommandLine values from the above example, which would be done like this: `event['MachineName']` and `event['CommandLine']`.

Implementing Sigma was achieved by replacing the rule parser that previously parsed SEC rules, and use a YAML library instead. Most of the work required to make these YAML function was spent on implementing the *condition* block from the Sigma specification [109]. One benefit with the new format, is that since the *selection* block-items are *AND*ed together, we are able to much quicker decide if a rule is applicable for a event, without having to iterate over every single condition in the rule. For testing this implementation, we used the rule found in Appendix F. The end architecture is less complex when compared to the one presented in Section 4.3. A figure representing the architecture for this iteration can be seen in Figure 4.2.

All implemented code will be available from the authors GitHub [122] after delivering this thesis.

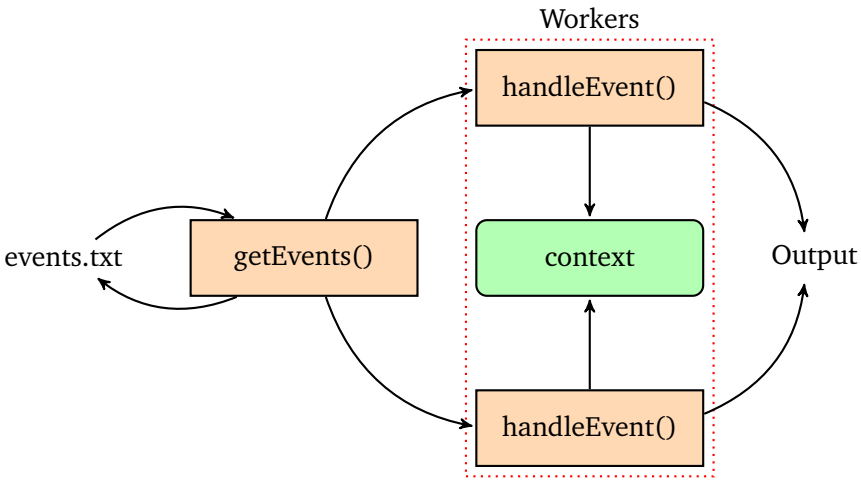


Figure 4.2: Second implementation in Go

Chapter 5

Results

In this chapter we will present the results from our experiments. Further analysis will be conducted in Chapter 6. We will denote our first implementation described in Section 4.3 with **MEC**, and the second implementation described in Section 4.4 as **MEC2**.

5.1 Dataset analysis

As discussed in Section 4.2, we wanted to analyze the datasets to get an impression if the frequency of events are relatively stable, or of there are any peaks in the dataset. In total there are **8** users and **5** hosts present in the dataset spanning both subsets. In the bar graphs in this section, the x axis represents time in 10 second intervals, and the y-axis represents the number of events during those 10 second intervals.
Figure 5.1 shows the data from the first subset. The data spans a time period of 76 minutes in total. As we can see, there are occasional spikes of events up to around 1500 and 2400 events. There is an average of **144** events per 10 second intervals. It is clear from the figure that there is always some background noise in the dataset. This is expected, as Windows event logs are fairly verbose.
In Figure 5.2 what immediately sticks out is the huge outlier with almost 25 000 events in a single 10 second interval. These events seems to be "Process Access"-events generated by a PowerShell-process enumerating all the processes on the system mutliple times. In Figure 5.3 we have removed that outlier to get a better view of the rest of the data in the graph. When we exclude the outlier, we get an average of **678** events per 10 second intervals. In addition to this, it is worth mentioning that the second subset shown in Figure 5.2 spans a lot shorter time period than the first subset, only roughly 12 minutes.

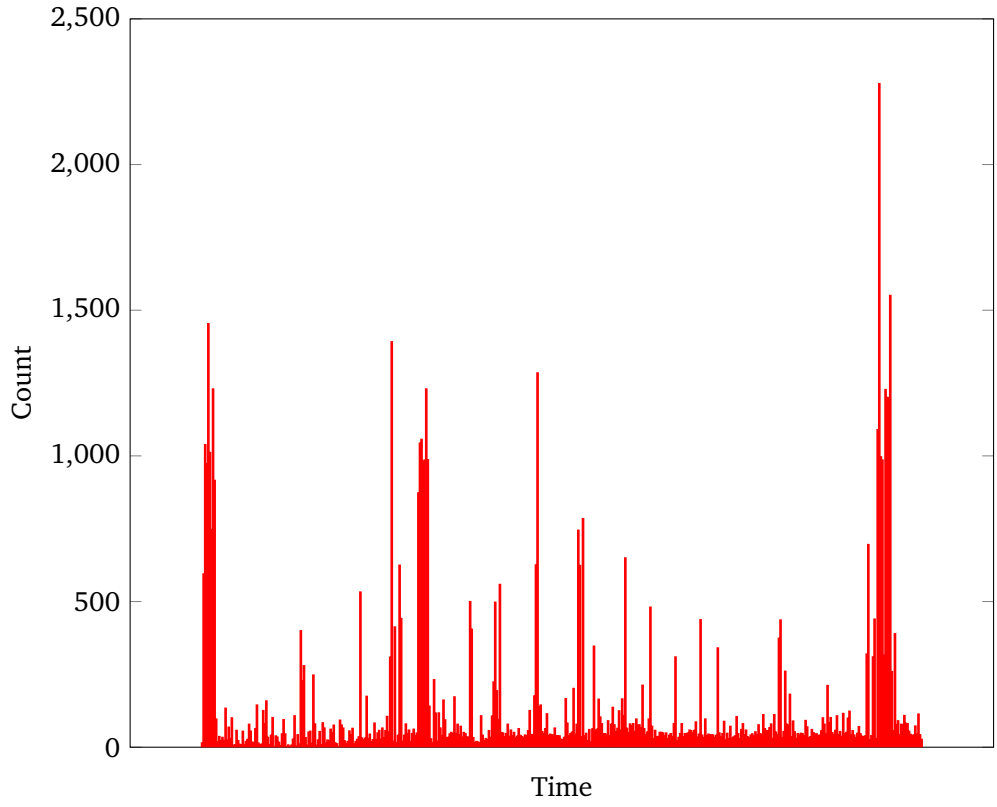


Figure 5.1: events in 10 sec intervals first subset

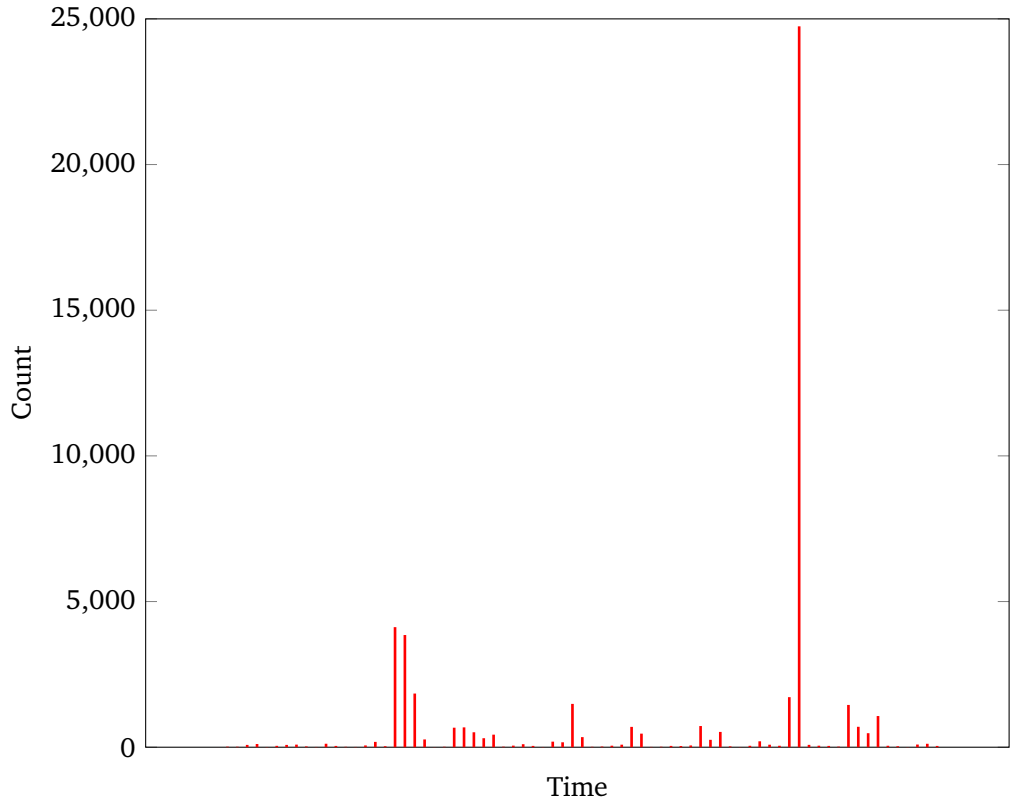


Figure 5.2: events in 10 sec intervals second subset

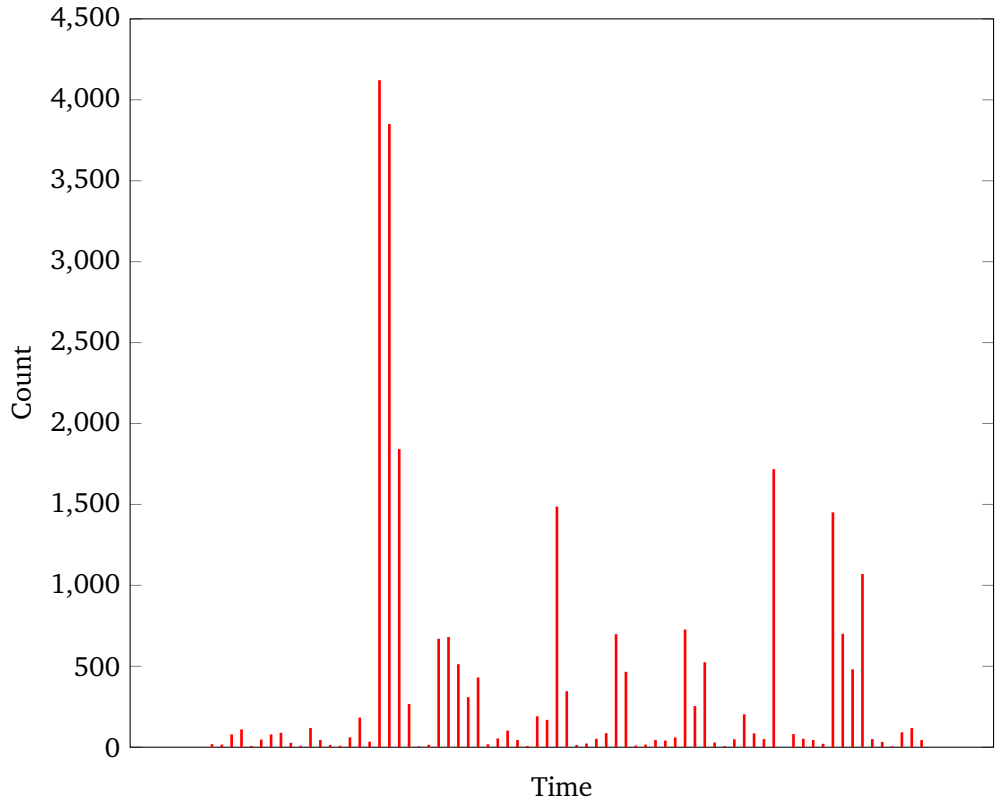


Figure 5.3: events in 10 sec intervals second subset with outlier removed

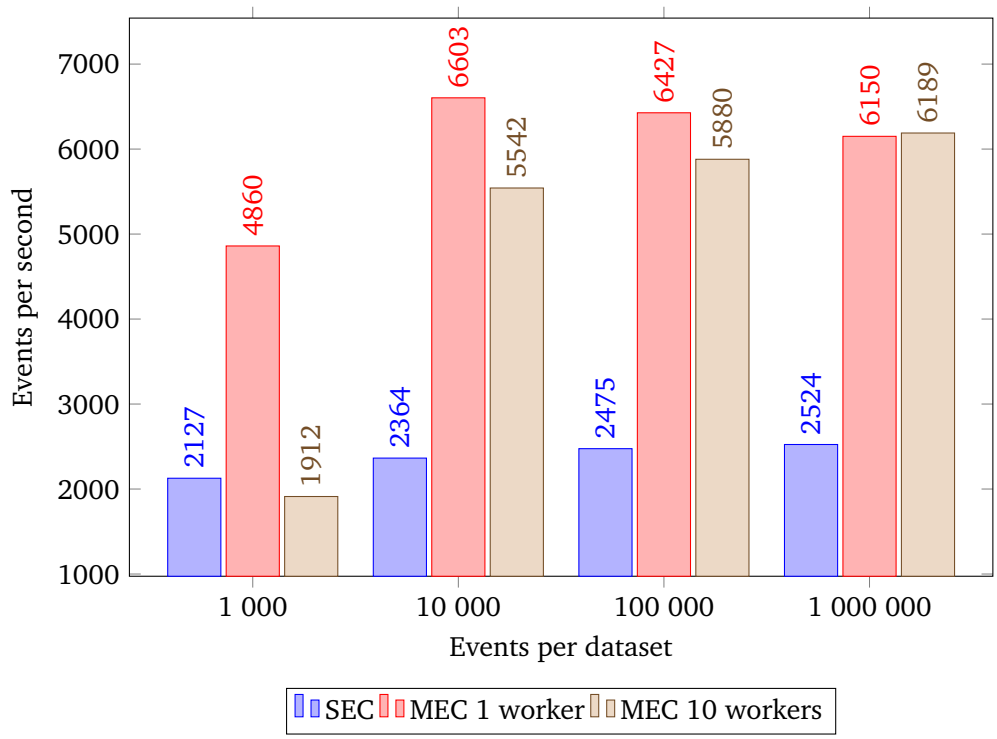


Figure 5.4: Baseline dataset

5.2 Implementation that uses SECs own regex-based rule format

Single core

When comparing our solution against SEC it makes sense to only use one thread for execution. As can be seen from Figure 5.4, MEC clearly outperforms SEC using both 1 and 10 workers. As can be seen from the figure, there is a certain disadvantage of running MEC with multiple workers when the dataset is small. We can attribute this to the additional overhead required by the Go runtime to control the goroutines in a single thread, and any locking that might occur between the goroutines against the rule context.
If we compare the baseline plot Figure 5.4 against the high signal, low noise dataset in Figure 5.5, we can clearly a speed improvement going from the baseline dataset, and over to the high signal, low noise dataset. The reason for this lies in the implementation of SEC and MEC. If we are able to match a rule quickly, we do not have to check all the other rules for a match, which when it adds up, saves some time and improves the overall throughput.

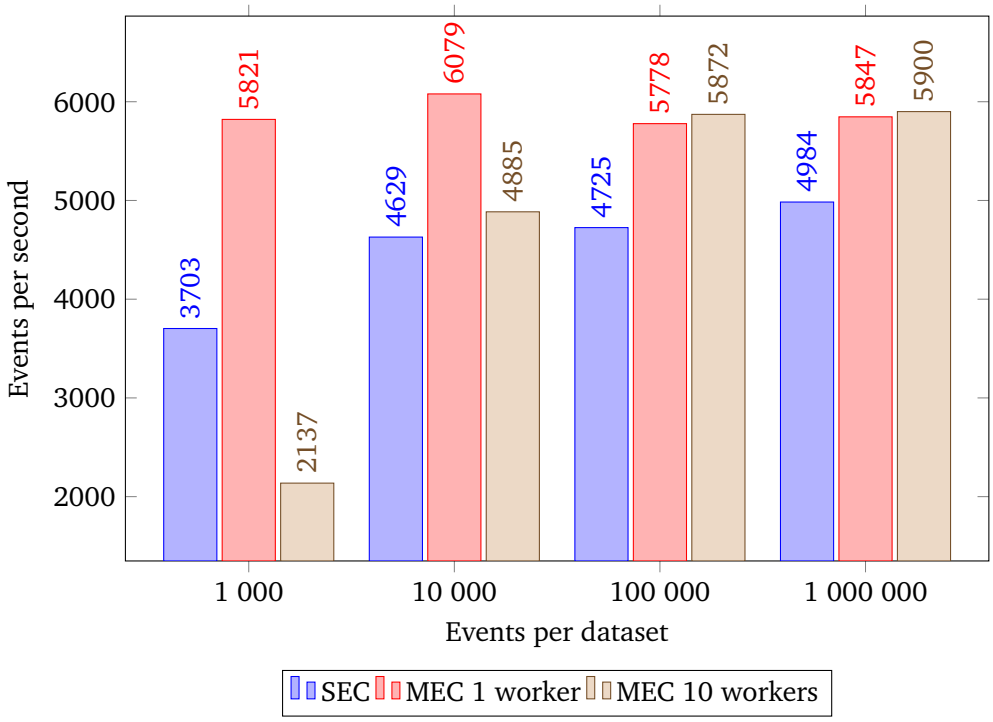


Figure 5.5: High signal, low noise dataset

Multiple cores

By using all the CPU cores available (4) instead of a single one, we can take better advantage of Gos concurrency model, and raise the throughput when using multiple workers and CPUs as seen in Figure 5.6 and Figure 5.7.
It is interesting to note the measurements for "1CPU,10W", which "catches up" with the other measurements at around 100 000 events in both Figure 5.6 and Figure 5.7. This is pretty much the same as what we saw in the single core test when we ran with 10 workers on a single thread. The time used to spin up the 10 workers is only outweighed at approximately 100 000 events.
As we can see from Figure 5.6 and Figure 5.7, the results of 1CPU,1W, 1CPU,10W and 4CPU,1W are generally performing the same. This is because they in general are the same. When we are limiting our script to 1 worker, it doesn't really matter how many cores we use, as only one core will be running the worker regardless. Likewise, when we are limited to only one CPU, spawning multiple workers only add additional overhead without any gains.
There is however a slight benefit to the 4CPU,1W when we consider the smallest dataset. This is because of the main-function in Go itself being a goroutine, so when we are creating a worker in another core, the main-function can work uninterrupted with reading the log files, while the worker is not blocking since it is running in another core.

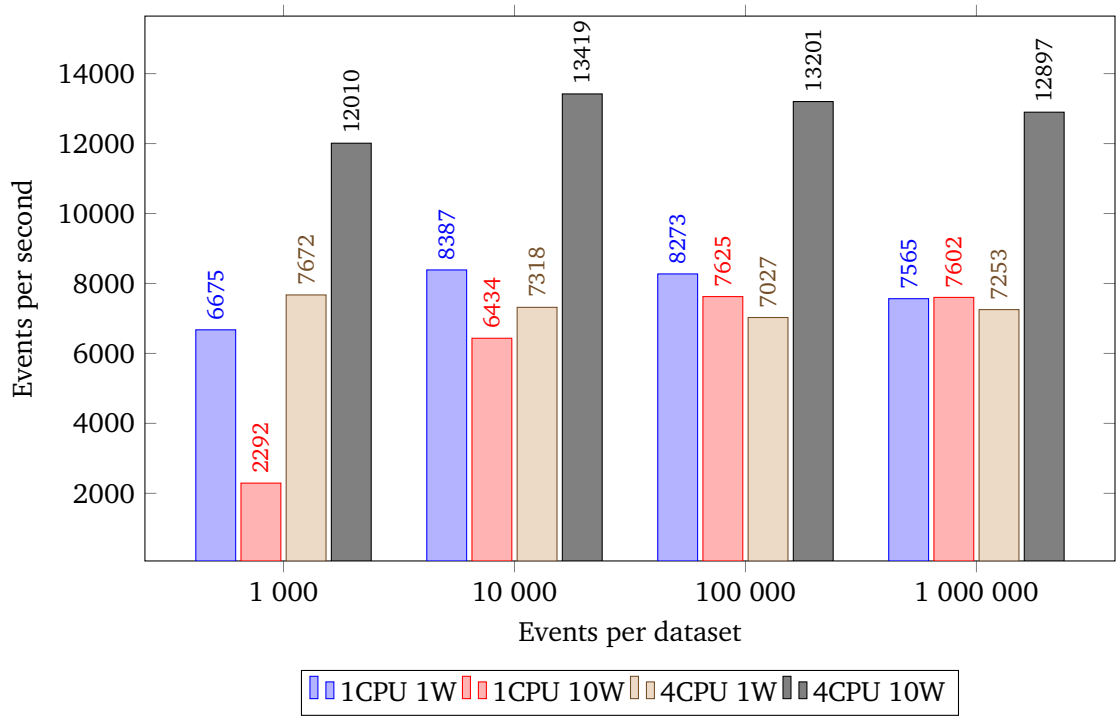


Figure 5.6: Concurrency with high signal low noise dataset

5.3 Implemented a new rule format

We are interested in seeing if there are any performance benefits from running our new rule implementation versus the re-implementation of the SEC rules. In Figure 5.8 we are running with only a single rule, using our high signal, low noise dataset. If we compare this to the concurrent version of MEC in Figure 5.6, we see a drastic improvement between the two.
The slight benefit to the 4CPU,1W we saw in Section 5.2, has changed drastically to become the next-best performing metric after implementing our new rule format.In addition there is now a larger separation between 1CPU,1W and 1CPU,10W as compared to the results in Section 5.2. This variation can again be explained by the fact that too many workers can be counter-productive, as they are blocking on the rule context between them. This makes the 1CPU,1W quicker, as there is not locking involved.
As stated in Section 4.2, we generated 1 000 rules randomly, to understand how multiple rules might impact performance. We ran it against our high signal, low noise dataset using 4 CPUs and 10 workers. The result can be found in Figure 5.9. As the reader can deduce, there is a drastic fall in events processed per second, because of the need to iterate over more rules.

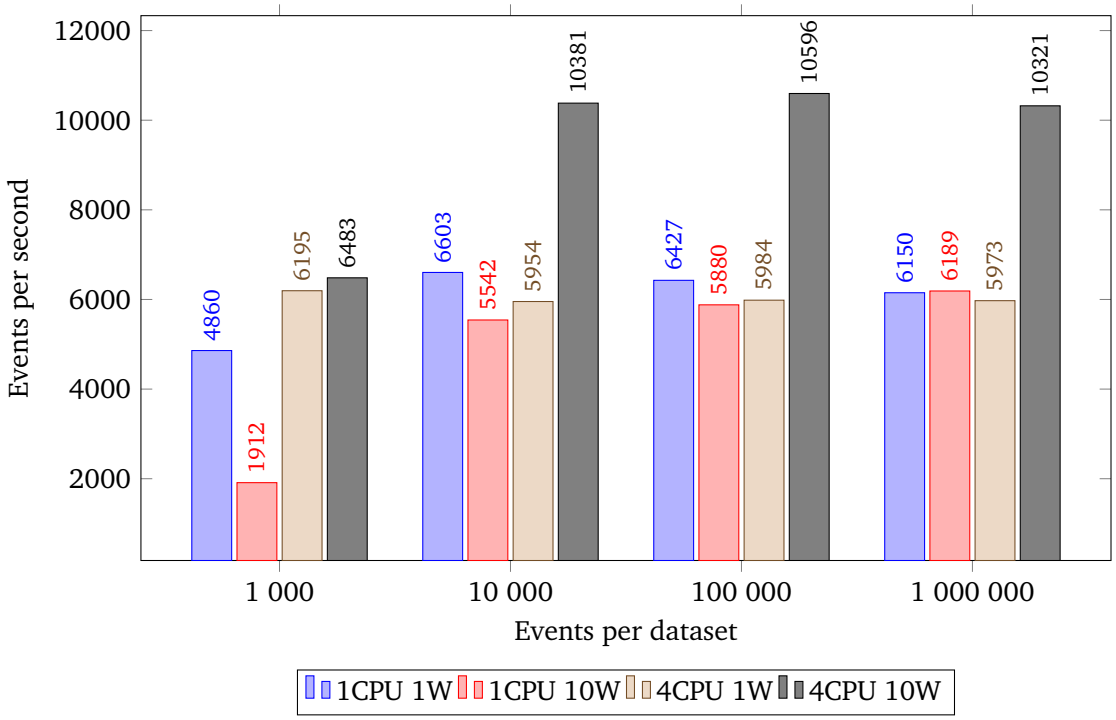


Figure 5.7: Concurrency with baseline dataset

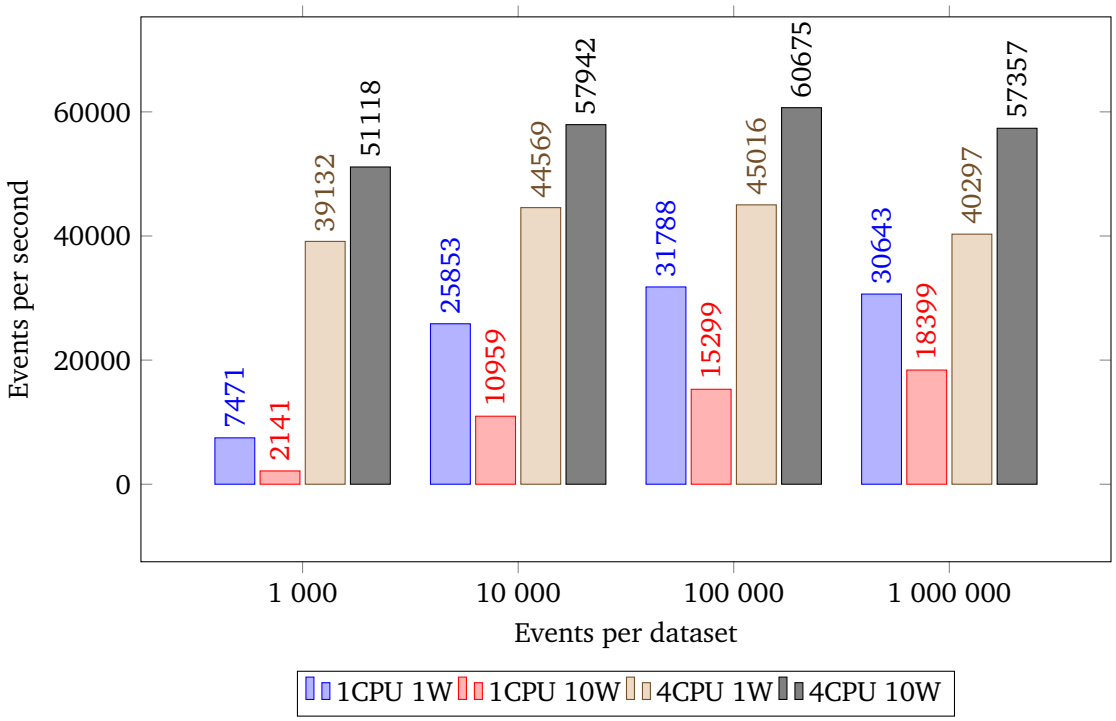


Figure 5.8: MEC2 concurrency with high signal low noise dataset

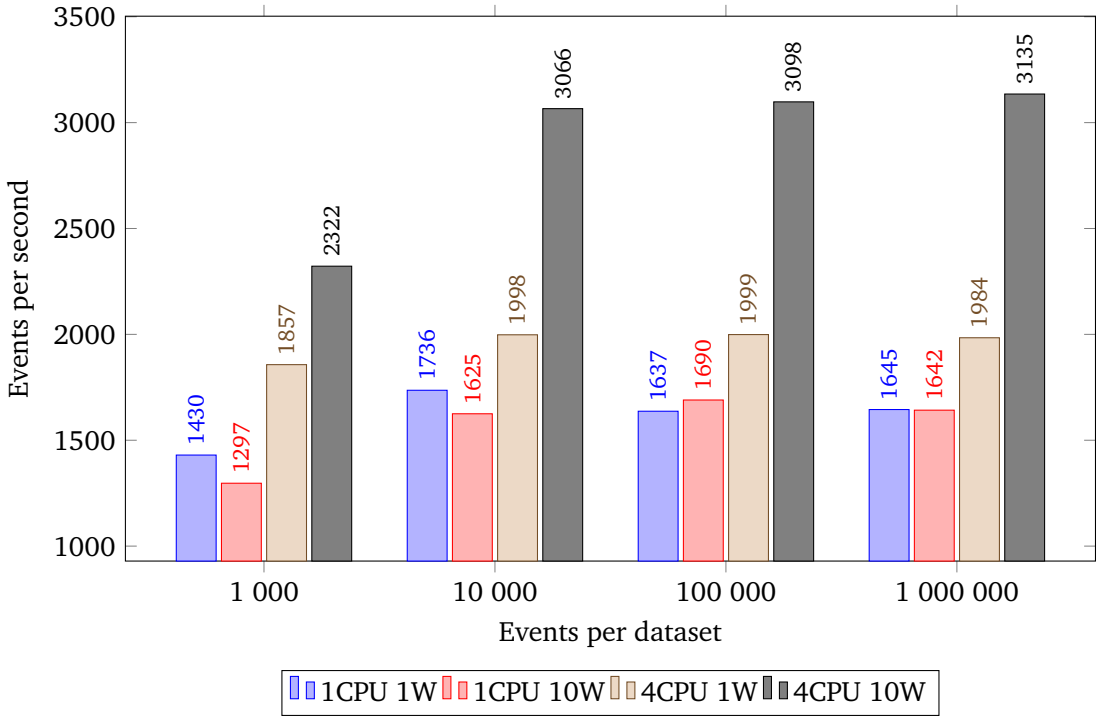


Figure 5.9: MEC2 1000 rules, high signal low noise dataset

Chapter 6

Discussion

In this chapter, we will discuss the results of our experiments, and how they line up with our research questions posed in Chapter 1. We will also outline any future work. This chapter provides a discussion of what implications the results of the experiments has, and presents different aspects of the work conducted.
The first research question regarding the state of the art in event correlation has been addressed in Chapter 2 where we have highlighted relevant studies and options for doing event correlation. We highlighted several different methods for doing event correlation.
The experiments conducted in this thesis evaluated a subset of possible features that might improve the performance of real time event correlation. We chose to compare our solution against SEC, as that seems to be the most popular open-source software for rule-based event correlation and used in a wide variety of sectors.
First of all we will extrapolate some numbers from the Mordor datasets that we used as explained in Section 3.1. As discussed in Section 5.1, the second scenario had an average events per 10 seconds of 678. This gives us 67.8 events per second. If we consider that the dataset contained 8 users and 5 hosts, we can try to make some assumptions regarding real world environments. If we consider an environment with 100 hosts, that would give us a ballpark estimation of 1356 events per second. If we consider an environment with 500 hosts, that brings our estimation to 6780 events per second. This is not taking into account any peaks in the data. If we consider the highest peak in the first scenario, as seen in Figure 5.1. Given a network size of 500 hosts, that would give us a peak at about 22 800 events per second. Now, that is probably unrealistic, as not all the hosts in the network would peak at the same time, producing massive amount of logs.
We have considered multiple ways that we could improve the way real time event correlation is done for Windows Event Logs in Chapter 3. As outlined in Chapter 4, we re-implemented what we considered the most important parts of SEC in Go, taking advantage of Go being a compiled program as discussed in Section 4.3. As seen in Figure 5.4 and Figure 5.5 just by re-implementing SEC alone saw performance improvements.

When comparing our solution against SEC it makes sense to only use one thread for execution. As can be seen from Figure 5.4, MEC clearly outperforms SEC using both 1 and 10 workers. As can be seen from the figure, there is a certain disadvantage of running MEC with multiple workers when the dataset is small. We can attribute this to the additional overhead required by the Go runtime to control the goroutines in a single thread, and any locking that might occur between the goroutines against the rule context.
If we compare the baseline plot Figure 5.4 against the high signal, low noise dataset in Figure 5.5, we can clearly a speed improvement going from the baseline dataset, and over to the high signal, low noise dataset. The reason for this lies in the implementation of SEC and MEC. If we are able to match a rule quickly, we do not have to check all the other rules for a match, which when it adds up, saves some time and improves the overall throughput.
Running this using equal conditions like the same dataset, and only a single core, we were able to outperform SEC with 20-40% using the high signal low noise dataset, and up to 89-135% when comparing with the baseline dataset. This clearly shows the benefits of utilizing a compiled language when performance is an important criteria. As discussed in Section 4.3, we wanted to add concurrency and threading to our solution, which allowed us to utilize the full capacity of the processor. As seen in Figure 5.6 and Figure 5.7, these improvements showed and greater event throughput. By taking full advantage of the system hardware by using all cores available to use. This gave us an even bigger increase in throughput compared to both SEC and our own implementation using only a single core. We saw performance improvements of 59-80% comparing our multi-threaded version to our single core version using the high signal low noise dataset, and improvements of 33-68% when using the baseline dataset.
In addition, we implemented a better time management system that extracts the UTC timestamp from the event, and uses that for the time-based correlation as opposed to SEC which uses the time of when SEC reads the log line from input. The difference here does not play a role processing-wise, as the timestamps in the datasets are set to a single point in time, which replicates how SEC works in our new solution. In a real world scenario this would not be the case, and we consider our solution to be a better implementation than the one used in SEC.
We also implemented a new way to pre-handle event logs when ingesting. We called this tokenizing, Section 3.2.5 and along with using Sigma Section 2.4.2 which reduced the reliance on regular expressions, as a new rule format, we were able to increase the throughput even further, as seen by Figure 5.8. This shows that we were not only able to improve the way real time event correlation is done for Windows Event Logs, but also show that our improvements give significant performance benefits.
The slight benefit to the 4CPU,1W we saw in Section 5.2, has changed drastically to become the next-best performing metric after implementing our new rule format. We attribute this again to the fact that the main-function in Go itself is a goroutine, so when we are creating a worker in another core, the main-function

can work uninterrupted with reading the log files, while the worker is not blocking on the rule context since it is running in another thread. In addition there is now a larger separation between 1CPU,1W and 1CPU,10W as compared to the results in Section 5.2. This variation can again be explained by the fact that too many workers can be counter-productive, as they are blocking on the rule context between them. This makes the 1CPU,1W quicker, as there is not locking involved. In our testing we primarily used 1 rule, which is unrealistic in a enterprise environment. To address this, we generated 1000 rules as explained in Section 4.2 and used that for testing performance as well. As can be seen in Figure 5.9, the solution takes a clear hit when having to parse a much larger number of rules, averaging at around 1600 events per second when ran against our high signal, low noise dataset using 4 CPUs and 10 workers. As discussed earlier in this chapter, that would still be within the threshold for an environment consisting of 100 hosts, but additional scaling would have to be done to support a larger number of events per second.

6.1 Future work

While we propose that there would be not have to be done a lot of changes to the current solution to implement other log formats in Section 3.2.6, we considered that future work for the single reason that we did not consider it necessary to address our research questions. However, it would be very interesting to see future work that covers correlating different log sources from for example network monitoring, application logs, etc.
We chose not to implement any form of output modularity as our focus was on increasing the performance of our solution. However it would be nice to be able to create granular output rules that takes some decision based on the alert severity and sends the alert via e-mail, instant messaging platforms or ticketing systems.
We discussed distributed correlation in Section 3.2.8 which would be beneficial for the redundancy, geolocation and scalability of the system. The implementation show in this thesis is primarily built to scale vertically, and interesting future work would be to add horizontal scaling to the proposed solution in this thesis, much like what is proposed in Figure 2.15, and tackle the challenges associated with load balancing, shared "context memory" between the correlators, and other possible obstacles.
We did unfortunately not have the ability to run our solution in any production environment, which would be of interest to prove the real-world use of our solution.
Lastly, as explained in Section 4.3.2 we did not achieve feature parity with SEC or Sigma in our implementation. We highlighted which parts we found interesting and necessary for this thesis to properly test and answer our research questions. However, it would still be interesting to see a complete implementation of SEC using a compiled language, in addition to fully integrating Sigma into MEC2, to allow for broader correlation actions using the various features in the rule spe-

cification.

Chapter 7

Conclusion

The primary contribution of this project is an improved method for correlating Windows Event Logs in time, in near real time. The goals of this thesis was to outlined the state of the art in real time event correlation, and implemented a solution that improves the way real time event correlation can be done with regards to Windows Event log correlation. We chose to compare our solution against SEC, as that seemed to be the most popular open-source software for rule-based event correlation and used in a wide variety of sectors as explained in Section 2.3. First of all we did a deep dive into the state of the art and considered several relevant types of event correlation. Rule-based event correlation was chosen because of its popular use in the security industry, and SEC was identified as the primary target that we wanted to compare our solution against.
A implementation was created that utilized the same rule-set as SEC. Just by using a compiled language like Go instead of a interpreted language like Perl, we saw improvements to the event throughput. When we implemented multi-threading and utilized the full processing power available to us on the test machine, we saw an even greater effect.
A new implementation was then proposed that uses different rules for correlating events and a different way to pre-process the events when ingesting. We considered the Sigma Section 2.4.2 rule format, and utilized tokenization Section 3.2.5 for making it easier to parse the event logs internally in our solution.
The experiments and the associated results present the event processing and correlation throughout which showed a varying level of increased performance, depending on the dataset and methods used for context management. We were able to outperform SEC with 20-40% using the high signal low noise dataset, and up to 89-135% when comparing with the baseline dataset. By taking full advantage of the system hardware by using all cores available to use and improving our rule format and internal representation of events gave us an even bigger increase in throughput compared to both SEC and our own implementation using only a single core. We saw performance improvements of 59-80% comparing our multi-threaded version to our single core version using the high signal low noise dataset, and improvements of 33-68% when using the baseline dataset.

Furthermore, we made an important contribution by implemented a better time management system that extracts the time from the event, and uses that for the time-based correlation as opposed to SEC which uses the time of when SEC reads the log line from input.
In conclusion, this thesis has outlined the state of the art in real time event correlation, and implemented a solution that improves the way real time event correlation can be done with regards to Windows Event log correlation and performs very good compared to SEC. Different implementations have been created and tested for performance through experiments using datasets that are both realistic, and optimized for testing performance. The experiments served as proof-of-concept that we were able to enhance and improve the event processing throughput and correctness compared to existing solutions. As a result, this thesis has made a contribution to event correlation, and more specifically for correlating Windows Event logs in near real time.

Bibliography

[1] AV-TEST, *The AV-TEST Security Report 2018/2019*, https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2018-2019.pdf, (Accessed on 05/16/2020), Jun. 2019.

[2] FireEye Mandiant Services, *M-Trends 2020*, https://www.fireeye.com/content/dam/collateral/en/mtrends-2020.pdf, (Accessed on 05/25/2020), 2020.

[3] M. Liu, Z. Xue, X. Xu, C. Zhong and J. Chen, 'Host-based intrusion detection system with system calls: Review and future trends', *ACM Comput. Surv.*, vol. 51, no. 5, Nov. 2018, ISSN: 0360-0300. DOI: 10.1145/3214304. [Online]. Available: https://doi.org/10.1145/3214304.

[4] A. Kramer and K. Z. Kramer, 'The potential impact of the covid-19 pandemic on occupational status, work from home, and occupational mobility', *Journal of Vocational Behavior*, p. 103 442, 2020, ISSN: 0001-8791. DOI: https://doi.org/10.1016/j.jvb.2020.103442. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0001879120300671.

[5] M. Brattstrom and P. Morreale, 'Scalable agentless cloud network monitoring', in *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, 2017, pp. 171–176.

[6] M. R. Fatemi and A. A. Ghorbani, 'Threat hunting in windows using big security log data', in *Security, Privacy, and Forensics Issues in Big Data*, IGI Global, 2020, pp. 168–188.

[7] Splunk, *Siem, aiops, application management, log management, machine learning, and compliance | splunk*, https://www.splunk.com/, (Accessed on 05/28/2020).

[8] QRadar, *Ibm qradar security intelligence | ibm*, https://www.ibm.com/security/security-intelligence/qradar, (Accessed on 05/28/2020).

[9] RSA NetWitness, *Rsa netwitness - threat detection and response*, https://www.rsa.com/en-us/products/threat-detection-response, (Accessed on 05/28/2020).

[10] M. Landauer, F. Skopik, M. Wurzenberger and A. Rauber, 'System log clustering approaches for cyber security applications: A survey', *Computers & Security*, vol. 92, p. 101 739, 2020, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2020.101739. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404820300250.

[11] OSSIM, *Ossim: The open source siem | alienvault*, https://cybersecurity.att.com/products/ossim, (Accessed on 05/28/2020).

[12] OSSEC, *Ossec - world's most widely used host intrusion detection system - hids*, https://www.ossec.net/, (Accessed on 05/28/2020).

[13] SEC, *Sec - open source and platform independent event correlation tool*, https://simple-evcorr.github.io/, (Accessed on 05/28/2020).

[14] P. He, J. Zhu, Z. Zheng and M. R. Lyu, 'Drain: An online log parsing approach with fixed depth tree', in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 33–40.

[15] CEF, *Micro focus security arcsight - common event format. implementing arcsight common event format (cef). version 25*, https://community.microfocus.com/dcvta86296/attachments/dcvta86296/connector-documentation/1197/2/CommonEventFormatV25.pdf, (Accessed on 05/29/2020).

[16] LEEF, *Log event extended format (leef)*, https://www.ibm.com/support/knowledgecenter/SS42VS_DSM/com.ibm.dsm.doc/c_LEEF_Format_Guide_intro.html, (Accessed on 05/29/2020).

[17] CIM, *Cim | dmtf*, https://www.dmtf.org/standards/cim, (Accessed on 05/29/2020).

[18] IDMEF, 'The intrusion detection message exchange format (idmef)', RFC Editor, RFC 4765, Mar. 2007, http://www.rfc-editor.org/rfc/rfc4765.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4765.txt.

[19] P. He, J. Zhu, S. He, J. Li and M. R. Lyu, 'Towards automated log parsing for large-scale log data analysis', *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 931–944, 2018.

[20] W. Xu, L. Huang, A. Fox, D. Patterson and M. I. Jordan, 'Detecting large-scale system problems by mining console logs', in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 117–132.

[21] Q. Fu, J.-G. Lou, Y. Wang and J. Li, 'Execution anomaly detection in distributed systems through unstructured log analysis', in *2009 ninth IEEE international conference on data mining*, IEEE, 2009, pp. 149–158.

[22] S. He, J. Zhu, P. He and M. R. Lyu, 'Experience report: System log analysis for anomaly detection', in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2016, pp. 207–218.

[23] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan and M. D. Ernst, 'Leveraging existing instrumentation to automatically infer invariant-constrained models', in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 267–277.

[24] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan and P. Martin, 'Assisting developers of big data analytics applications when deploying on hadoop clouds', in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 402–411.

[25] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou and S. Savage, 'Be conservative: Enhancing failure diagnosis with proactive logging', in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 293–306.

[26] K. Nagaraj, C. Killian and J. Neville, 'Structured comparative analysis of systems logs to diagnose performance problems', in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 353–366.

[27] A. Oprea, Z. Li, T.-F. Yen, S. H. Chin and S. Alrwais, 'Detection of early-stage enterprise infection by mining large-scale log data', in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, IEEE, 2015, pp. 45–56.

[28] Z. Gu, K. Pei, Q. Wang, L. Si, X. Zhang and D. Xu, 'Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis', in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, IEEE, 2015, pp. 57–68.

[29] Ultimate Windows Security, *Randy's windows security log encyclopedia*, `https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/default.aspx`, (Accessed on 05/29/2020).

[30] A. Schuster, 'Introducing the microsoft vista event log file format', *Digital Investigation*, vol. 4, pp. 65–72, 2007, ISSN: 1742-2876. DOI: `https://doi.org/10.1016/j.diin.2007.06.015`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S1742287607000424`.

[31] Microsoft, *Windows event log - win32 apps | microsoft docs*, `https://docs.microsoft.com/en-us/windows/win32/wes/windows-event-log`, (Accessed on 05/29/2020).

[32] Microsoft, *Use windows event forwarding to help with intrusion detection (windows 10) - windows security | microsoft docs*, `https://docs.microsoft.com/en-us/windows/security/threat-protection/use-windows-event-forwarding-to-assist-in-intrusion-detection`, (Accessed on 05/29/2020).

[33] Splunk, *Monitor windows event log data - splunk documentation*, https://docs.splunk.com/Documentation/Splunk/8.0.3/Data/MonitorWindowseventlogdata, (Accessed on 05/29/2020).

[34] Winlogbeat, *Configure winlogbeat | winlogbeat reference [7.7] | elastic*, https://www.elastic.co/guide/en/beats/winlogbeat/current/configuration-winlogbeat-options.html, (Accessed on 05/29/2020).

[35] NXLog, *107.2. collecting event log data | log management solutions*, https://nxlog.co/documentation/nxlog-user-guide/eventlog-collecting.html, (Accessed on 05/29/2020).

[36] Sysmon, *Sysmon - windows sysinternals | microsoft docs*, https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon, (Accessed on 05/29/2020).

[37] A. Prokhorov, *Correlation (in statistics) - encyclopedia of mathematics*, https://encyclopediaofmath.org/wiki/Correlation_(in_statistics), (Accessed on 05/30/2020).

[38] Kent State University, *Pearson correlation - spss tutorials - libguides at kent state university*, https://libguides.library.kent.edu/SPSS/PearsonCorr, (Accessed on 05/30/2020).

[39] A. Prokhorov, *Spearman coefficient of rank correlation - encyclopedia of mathematics*, https://encyclopediaofmath.org/wiki/Spearman_coefficient_of_rank_correlation, (Accessed on 05/30/2020).

[40] A. Prokhorov, *Kendall coefficient of rank correlation - encyclopedia of mathematics*, https://encyclopediaofmath.org/wiki/Kendall_coefficient_of_rank_correlation, (Accessed on 05/30/2020).

[41] L. A. Goodman and W. H. Kruskal, 'Measures of association for cross classifications', *Journal of the American Statistical Association*, vol. 49, no. 268, pp. 732–764, 1954, ISSN: 01621459. [Online]. Available: http://www.jstor.org/stable/2281536.

[42] W. Wei, F. Chen, Y. Xia and G. Jin, 'A rank correlation based detection against distributed reflection dos attacks', *IEEE Communications Letters*, vol. 17, no. 1, pp. 173–175, 2013.

[43] G. Jiang and G. Cybenko, 'Temporal and spatial distributed event correlation for network security', in *Proceedings of the 2004 American Control Conference*, IEEE, vol. 2, 2004, pp. 996–1001.

[44] C. Kruegel, F. Valeur and G. Vigna, *Intrusion detection and correlation: challenges and solutions*. Springer Science & Business Media, 2004, vol. 14.

[45] R. M. Keller, 'Computer science: Abstraction to implementation', *Harvey Mudd College, Claremont, CA, United States*, 2001.

[46] A. Bouloutas, G. W. Hart and M. Schwartz, 'Simple finite-state fault detectors for communication networks', *IEEE Transactions on Communications*, vol. 40, no. 3, pp. 477–479, 1992.

[47] R. N. Cronk, P. H. Callahan and L. Bernstein, 'Rule-based expert systems for network management and operations: An introduction', *IEEE Network*, vol. 2, no. 5, pp. 7–21, 1988.

[48] D. M. Meira, 'A model for alarm correlation in telecommunications networks', *Computer Science Institute of Exact Sciences (ICEX) of the Federal University of Minas Gerais*, 1997.

[49] L. Lewis, 'A case-based reasoning approach to the management of faults in communication networks', in *IEEE INFOCOM '93 The Conference on Computer Communications, Proceedings*, 1993, 1422–1429 vol.3.

[50] Suricata, *Suricata | open source ids / ips / nsm engine*, `https://suricata-ids.org/`, (Accessed on 05/31/2020).

[51] Snort, *Snort - network intrusion detection & prevention system*, `https://www.snort.org/`, (Accessed on 05/31/2020).

[52] swatchdog, *Toddatkins/swatchdog: The simple log watcher formerly known as swatch.* `https://github.com/ToddAtkins/swatchdog`, (Accessed on 05/31/2020).

[53] K. Thompson, *Logsurfer*, Jun. 2017. [Online]. Available: `https://www.crypt.gen.nz/logsurfer/`.

[54] R. Vaarandi, 'Sec - a lightweight event correlation tool', in *IEEE Workshop on IP Operations and Management*, Oct. 2002, pp. 111–115. DOI: `10.1109/IPOM.2002.1045765`.

[55] Prelude, *Overview - prelude siem - unity 360*, `https://www.prelude-siem.org/`, (Accessed on 05/31/2020).

[56] Wazuh, *Wazuh - the open source security platform*, `https://wazuh.com/`, (Accessed on 05/31/2020).

[57] Apache Metron, *Apache metron big data security*, `https://metron.apache.org/`, (Accessed on 05/31/2020).

[58] MozDef, *Mozilla/mozdef: Mozdef: Mozilla enterprise defense platform*, `https://github.com/mozilla/MozDef`, (Accessed on 05/31/2020).

[59] OpenNMS, *The opennms group, inc.* `https://www.opennms.com/`, (Accessed on 05/31/2020).

[60] M. Kont, M. Pihelgas, K. Maennel, B. Blumbergs and T. Lepik, 'Frankenstack: Toward real-time red team feedback', in *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*, IEEE, 2017, pp. 400–405.

[61] M. Farshchi, 'Anomaly detection using logs and metrics analysis for system application operations', 2018.

[62] R. Vaarandi, 'A data clustering algorithm for mining patterns from event logs', in *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*, IEEE, 2003, pp. 119–126.

[63] A. Aamodt and E. Plaza, 'Case-based reasoning: Foundational issues, methodological variations, and system approaches', *AI Commun.*, vol. 7, no. 1, pp. 39–59, Mar. 1994, ISSN: 0921-7126.

[64] S. Slade, 'Case-based reasoning: A research paradigm', *AI magazine*, vol. 12, no. 1, pp. 42–42, 1991.

[65] T. Davies and S. Russell, 'A logical approach to reasoning by analogy.', Aug. 1987, pp. 264–270.

[66] D. B. Leake and R. F. Remindings, 'Cbr in context: The present and future', MIT Press, 1996, pp. 3–30.

[67] D. Schwartz, S. Stoecklin and E. Yilmaz, 'A case-based approach to network intrusion detection', Feb. 2002, 1084–1089 vol.2, ISBN: 0-9721844-1-4. DOI: 10.1109/ICIF.2002.1020933.

[68] S. Kapetanakis, A. Filippoupolitis, G. Loukas and T. S. A. Murayziq, 'Profiling cyber attackers using case-based reasoning', in *Nineteenth UK Workshop on Case-Based Reasoning (UK-CBR 2014)*, part of AI-2014 Thirty-fourth SGAI International Conference on Artificial Intelligence, Cambridge, UK 9-11 December 2014, CEUR, Dec. 2014. [Online]. Available: http://gala.gre.ac.uk/id/eprint/14950/.

[69] M. Han, H. Han, A. Kang, B. Kwak, A. Mohaisen and H. Kim, 'Whap: Web-hacking profiling using case-based reasoning', English, in *2016 IEEE Conference on Communications and Network Security, CNS 2016*, 2016 IEEE Conference on Communications and Network Security, CNS 2016 ; Conference date: 17-10-2016 Through 19-10-2016, Institute of Electrical and Electronics Engineers Inc., Feb. 2017, pp. 344–345. DOI: 10.1109/CNS.2016.7860503.

[70] G. Dodig-Crnkovic and A. Cicchetti, 'Computational aspects of model-based reasoning', in *Springer Handbook of Model-Based Science*, L. Magnani and T. Bertolotti, Eds. Cham: Springer International Publishing, 2017, pp. 695–718, ISBN: 978-3-319-30526-4. DOI: 10.1007/978-3-319-30526-4_32. [Online]. Available: https://doi.org/10.1007/978-3-319-30526-4_32.

[71] G. Jakobson and M. Weissman, 'Alarm correlation', *IEEE Network*, vol. 7, no. 6, pp. 52–59, 1993.

[72] S. Poll, D. Iverson, J. Ou, D. Sanderfer and A. Patterson-Hine, 'System modeling and diagnostics for liquefying-fuel hybrid rockets', 2003.

[73] M. Steinder and A. S. Sethi, 'The present and future of event correlation: A need for end-to-end service fault localization', 2001.

[74] V. Venkatasubramanian, R. Rengaswamy, K. Yin and S. N. Kavuri, 'A review of process fault detection and diagnosis: Part i: Quantitative model-based methods', *Computers & Chemical Engineering*, vol. 27, no. 3, pp. 293–311, 2003, ISSN: 0098-1354. DOI: https://doi.org/10.1016/S0098-1354(02)00160-6. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0098135402001606.

[75] S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini and D. Ohsie, 'High speed and robust event correlation', *IEEE Communications Magazine*, vol. 34, no. 5, pp. 82–90, 1996.

[76] M. Gupta and M. Subramanian, 'Preprocessor algorithm for network management codebook.', in *Workshop on Intrusion Detection and Network Monitoring*, 1999, pp. 93–102.

[77] B. Gruschke, *Integrated event management: Event correlation using dependency graphs*, 1998.

[78] F. Kavousi and B. Akbari, 'A bayesian network-based approach for learning attack strategies from intrusion alerts', *Security and Communication Networks*, vol. 7, no. 5, pp. 833–853, 2014. DOI: 10.1002/sec.786. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.786. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.786.

[79] X. Qin and W. Lee, 'Attack plan recognition and prediction using causal networks', in *20th Annual Computer Security Applications Conference*, 2004, pp. 370–379.

[80] Y.-Y. Chen, Y.-H. Lin, C.-C. Kung, M.-H. Chung, I. Yen *et al.*, 'Design and implementation of cloud analytics-assisted smart power meters considering advanced artificial intelligence as edge analytics in demand-side management for smart homes', *Sensors*, vol. 19, no. 9, p. 2047, 2019.

[81] R. Lippmann, 'An introduction to computing with neural nets', *IEEE Assp magazine*, vol. 4, no. 2, pp. 4–22, 1987.

[82] F. Pouget and M. Dacier, 'Alert correlation: Review of the state of the art', *TechnicalReport EURECOM*, vol. 1271, 2003.

[83] I. Friedberg, F. Skopik, G. Settanni and R. Fiedler, 'Combating advanced persistent threats: From network event correlation to incident detection', *Computers & Security*, vol. 48, pp. 35–57, 2015.

[84] T.-C. Lin, C.-C. Guo and C.-S. Yang, 'Detecting advanced persistent threat malware using machine learning-based threat hunting', in *European Conference on Cyber Warfare and Security*, Academic Conferences International Limited, 2019, pp. 760–XX.

[85] H. Wietgrefe, K.-D. Tuchs, K. Jobmann, G. Carls, P. Fröhlich, W. Nejdl and S. Steinfeld, 'Using neural networks for alarm correlation in cellular phone networks', in *International Workshop on Applications of Neural Networks to Telecommunications (IWANNT)*, Citeseer, 1997, pp. 248–255.

[86] A. Hanemann and P. Marcu, 'Algorithm design and application of service-oriented event correlation', in *2008 3rd IEEE/IFIP International Workshop on Business-driven IT Management*, IEEE, 2008, pp. 61–70.

[87] S. Saad and I. Traore, 'Extracting attack scenarios using intrusion semantics', in *International Symposium on Foundations and Practice of Security*, Springer, 2012, pp. 278–292.

[88] M. Ficco *et al.*, 'Security event correlation approach for cloud computing.', *IJHPCN*, vol. 7, no. 3, pp. 173–185, 2013.

[89] L. Mé, E. Totel and B. Vivinis, 'A language driven intrusion detection system for event and alert correlation', *International Federation for Information Processing Digital Library; Security and Protection in Information Processing Systems;*, vol. 147, Aug. 2004. DOI: 10.1007/1-4020-8143-X_14.

[90] R. Vaarandi, *Tools and Techniques for Event Log Analysis*. Tallinn University of Technology Press, 2005.

[91] R. Vaarandi, 'Sec-a lightweight event correlation tool', in *IEEE Workshop on IP Operations and Management*, IEEE, 2002, pp. 111–115.

[92] R. Vaarandi, B. Blumbergs and E. Çalışkan, 'Simple event correlator-best practices for creating scalable configurations', in *2015 IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision*, IEEE, 2015, pp. 96–100.

[93] D. Lang, 'Building a 100k log/sec logging infrastructure', in *Presented as part of the 26th Large Installation System Administration Conference (LISA 12)*, San Diego, CA: USENIX, 2012, pp. 203–213. [Online]. Available: https://www.usenix.org/conference/lisa12/technical-sessions/presentation/lang_david.

[94] memcached, *Memcached - a distributed memory object caching system*, https://memcached.org/, (Accessed on 05/31/2020).

[95] OSSEC, *Rules syntax — ossec*, https://ossec-docs.readthedocs.io/en/latest/docs/syntax/head_rules.html, (Accessed on 05/31/2020).

[96] AlienVault, *Working with alienvault hids rules*, https://cybersecurity.att.com/documentation/usm-appliance/ids-configuration/working-with-alienvault-hids-rules.htm, (Accessed on 05/31/2020).

[97] Neo23x0, *Neo23x0/sigma: Generic signature format for siem systems*, https://github.com/Neo23x0/sigma, (Accessed on 05/31/2020).

[98] EsperTech, *Esper - espertech*, http://www.espertech.com/esper/, (Accessed on 05/31/2020).

[99] EQL, *Basic syntax — eql 0.9.2 documentation*, https://eql.readthedocs.io/en/latest/query-guide/basic-syntax.html, (Accessed on 05/31/2020).

[100] C. L. Forgy, 'Rete: A fast algorithm for the many pattern/many object pattern match problem', in *Readings in Artificial Intelligence and Databases*, Elsevier, 1989, pp. 547–559.

[101] R. B. Doorenbos, 'Production matching for large learning systems.', CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, Tech. Rep., 1995.

[102] Razorbliss, *File:rete.svg — Wikipedia, the free encyclopedia*, [Online; accessed 31-May-2020], 2011. [Online]. Available: https://commons.wikimedia.org/wiki/File:Rete.svg.

[103] J. P. Rouillard, 'Real-time log file analysis using the simple event correlator (sec).', in *LISA*, vol. 4, 2004, pp. 133–150.

[104] MITRE, *Car-2013-04-002: Quick execution of a series of suspicious commands | mitre cyber analytics repository*, https://car.mitre.org/analytics/CAR-2013-04-002/, (Accessed on 06/01/2020).

[105] SEC, *Man page of sec*, https://simple-evcorr.github.io/man.html, (Accessed on 06/01/2020).

[106] Elastic, *Elastic stack: Elasticsearch, kibana, beats & logstash | elastic*, https://www.elastic.co/elastic-stack, (Accessed on 06/01/2020).

[107] YAML, *The official yaml web site*, https://yaml.org/, (Accessed on 06/01/2020).

[108] Sigma, *Sigma/rules at master - neo23x0/sigma*, https://github.com/Neo23x0/sigma/tree/master/rules, (Accessed on 06/01/2020).

[109] *Specification · neo23x0/sigma wiki*, https://github.com/Neo23x0/sigma/wiki/Specification, (Accessed on 06/01/2020).

[110] Splunk, *Splunk/securitydatasets: Home for splunk security datasets.* https://github.com/splunk/securitydatasets, (Accessed on 06/01/2020).

[111] Hunters Forge, *Hunters-forge/mordor: Re-play adversarial techniques*, https://github.com/hunters-forge/mordor, (Accessed on 06/01/2020).

[112] MITRE, *Mitre att&ck evaluations*, https://attackevals.mitre.org/, (Accessed on 06/01/2020).

[113] MITRE, *Mitre att&ck®*, https://attack.mitre.org/, (Accessed on 06/01/2020).

[114] *Mitre att&ck® evaluations*, https://attackevals.mitre.org/APT3/, (Accessed on 06/01/2020).

[115] E. L. Barse, H. Kvarnstrom and E. Jonsson, 'Synthesizing test data for fraud detection systems', in *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, IEEE, 2003, pp. 384–394.

[116] L. Wall *et al.*, *The perl programming language*, 1994.

[117] Go, *The go programming language*, `https://golang.org/`, (Accessed on 06/01/2020).

[118] Rust, *Rust programming language*, `https://www.rust-lang.org/`, (Accessed on 06/01/2020).

[119] L. Henriques and J. Bernardino, 'Performance of memory deallocation in c++, c# and java', 2018.

[120] Go, *The go memory model - the go programming language*, `https://golang.org/ref/mem`, (Accessed on 06/01/2020), May 2014.

[121] SEC, *Simple-evcorr/rulesets: Simple event correlator ruleset repository*, `https://github.com/simple-evcorr/rulesets/`, (Accessed on 06/01/2020).

[122] M. Ingesen, *Martiningesen/master-thesis*, `https://github.com/MartinIngesen/master-thesis`, (Accessed on 06/02/2020), Jun. 2020.

Appendix A

Sysmon to Syslog Python script

Code listing A.1: Sysmon to Syslog Python script

```
import json

def convertEvents(sysmon):
    for event in sysmon:
        if "Microsoft-Windows-Sysmon" in event:
            event = json.loads(event)

            m = event["message"]
            m = m.replace("\n", "␣␣")

            if "computer_name" in event:
                h = event["computer_name"]
            elif "winlog" in event:
                h = event["winlog"]["computer_name"]
            else:
                h = "NOHOSTNAME"

            x = f"<14>Jan␣01␣00:00:00␣{h}␣Microsoft-Windows-Sysmon[2092]:␣{m}"
            print(x)

with open('./caldera_attack_evals_round1_day1_2019-10-20201108.json','r') as sysmon:
    convertEvents(sysmon)

with open('./empire_apt3_2019-05-14223117.json','r') as sysmon:
    convertEvents(sysmon)
```

Appendix B

Extracting events in 10s intervals

Code listing B.1: Extracting events in 10s intervals

```
import json
from datetime import datetime

epoch = datetime.utcfromtimestamp(0)

depth = 9 # 10s intervals
m = {}

def unix_time_millis(datetime):
    return str((datetime - epoch).total_seconds() * 1000.0).replace(".0", "")

def convertEvents(sysmon):
    for event in sysmon:
        if "Microsoft-Windows-Sysmon" in event:
            event = json.loads(event)
            timestamp = event['@timestamp']
            parsed = datetime.strptime(timestamp,"%Y-%m-%dT%H:%M:%S.%fZ")
            millis = unix_time_millis(parsed)
            top = millis[:depth]

            if top in m:
                m[top] += 1
            else:
                m[top] = 1

with open('./caldera_attack_evals_round1_day1_2019-10-20201108.json', 'r') as sysmon:
    convertEvents(sysmon)

with open('./empire_apt3_2019-05-14223117.json', 'r') as sysmon:
    convertEvents(sysmon)

for x in m:
    print(f"({x},{m[x]})")
```

Appendix C

Extracting users from dataset

Code listing C.1: Extracting users from dataset

```
import json

users = {}

def convertEvents(sysmon):
    for event in sysmon:
        if "Microsoft-Windows-Sysmon" in event:
            event = json.loads(event)

            if "winlog" in event:
                if "event_data" in event["winlog"]:
                    if "User" in event["winlog"]["event_data"]:
                        user = event["winlog"]["event_data"]["User"]
                        if user not in m:
                            users[user] = 1

with open('./caldera_attack_evals_round1_day1_2019-10-20201108.json', 'r') as sysmon:
    convertEvents(sysmon)

with open('./empire_apt3_2019-05-14223117.json', 'r') as sysmon:
    convertEvents(sysmon)

print(f"There␣are␣{len(users)}␣in␣total:")
for user in users:
    print(user)
```

Appendix D

Extracting computers from dataset

Code listing D.1: Extracting computers from dataset

```
import json

computers = {}

def convertEvents(sysmon):
    for event in sysmon:
        if "Microsoft-Windows-Sysmon" in event:
            event = json.loads(event)

            if "computer_name" in event:
                hostname = event["computer_name"]
            elif "winlog" in event:
                hostname = event["winlog"]["computer_name"]

            if hostname not in computers:
                computers[hostname] = 1

with open('./caldera_attack_evals_round1_day1_2019-10-20201108.json', 'r') as sysmon:
    convertEvents(sysmon)

with open('./empire_apt3_2019-05-14223117.json', 'r') as sysmon:
    convertEvents(sysmon)

print(f"There␣are␣{len(computers)}␣in␣total:")
for computer in computers:
    print(computer)
```

Appendix E

SEC rule used in testing

Code listing E.1: SEC rule used in testing

```
# whoami
# $1 - hostname
# $2 - executable
# $3 - username
type=Single
ptype=RegExp
pattern=<\d+>\S+\s+\d+\s\d\d:\d\d:\d\d\s(\S+).*Process Create.*OriginalFileName:
\s+((?i)whoami.exe).*User: (\S+)
desc=$0
action=event CAR-2013-04-002_for_$3_on_$1

# quser
type=Single
ptype=RegExp
pattern=<\d+>\S+\s+\d+\s\d\d:\d\d:\d\d\s(\S+).*Process Create.*OriginalFileName:
\s+((?i)quser.exe).*User: (\S+)
desc=$0
action=event CAR-2013-04-002_for_$3_on_$1

# hostname
type=Single
ptype=RegExp
pattern=<\d+>\S+\s+\d+\s\d\d:\d\d:\d\d\s(\S+).*Process Create.*OriginalFileName:
\s+((?i)hostname.exe).*User: (\S+)
desc=$0
action=event CAR-2013-04-002_for_$3_on_$1

# collector
# $1 - username
# $2 - hostname
type=SingleWithThreshold
ptype=RegExp
pattern=CAR-2013-04-002_for_(\S+)_on_(\S+)
desc=$0
action=write - CAR-2013-04-002: Quick execution of a series of suspicious commands
    detected on host $2 from user $1
window=10
thresh=3

#
```

```
# SEC Performance Test Rule
# Look for EOF at the end of the line, and send ourselves
# a USR1 signal to dump statistics, and a TERM signal to
#end the program.
type=Single
ptype=RegExp
pattern=EOF\s*$
desc=$0
action=eval %k ( $pid=$$$; kill(TERM, $pid));
```

Appendix F

Sigma rule used in testing

Code listing F.1: Sigma rule used in testing

```
title: Quick Execution of a Series of Suspicious Commands
id: 61ab5496-748e-4818-a92f-de78e20fe1f1
description: Detects multiple suspicious process in a limited timeframe
logsource:
    category: process_creation
    product: windows
detection:
    selection:
        CommandLine:
            - whoami
            - quser
            - hostname
    timeframe: 10s
    condition: selection | count() by MachineName >= 3
```

Appendix G

Rule generator

Code listing G.1: Rule generator

```
import uuid
import random

NUM_RULES = 1000
MEC_OUTPUT_FOLDER = "./output/"
MEC_SUFFIX = "_rule.yml"

SEC_OUTPUT_FOLDER = "./sec-output/"
SEC_SUFFIX = "_rule.sec"

mec_template = ""
sec_template = ""

COMMANDLINE = [
    "arp",
    "at",
    "whoami",
    "attrib",
    "cscript",
    "dsquery",
    "hostname",
    "ipconfig",
    "mimikatz",
    "nbstat",
    "net",
    "netsh",
    "nslookup",
    "ping",
    "quser",
    "qwinsta",
    "reg",
    "runas",
    "sc",
    "schtasks",
    "ssh",
    "systeminfo",
    "taskkill",
    "telnet",
    "tracert",
    "wscript",
```

```
    "xcopy"
]

with open('template.yml', 'r') as template_file:
    mec_template = template_file.read()

with open('template.sec', 'r') as template_file:
    sec_template = template_file.read()

for i in range(NUM_RULES):
    mt = mec_template
    st = sec_template
    mec_path = f"{MEC_OUTPUT_FOLDER}{i}{MEC_SUFFIX}"
    sec_path = f"{SEC_OUTPUT_FOLDER}{i}{SEC_SUFFIX}"

    RANDOM_ID = str(uuid.uuid4())
    TIMEFRAME = str(random.randrange(10,30))
    COUNT = str(random.randrange(3, 6))
    COMMAND_1 = str(random.choice(COMMANDLINE))
    COMMAND_2 = str(random.choice(COMMANDLINE))
    COMMAND_3 = str(random.choice(COMMANDLINE))

    mt = mt.replace("{RANDOM_ID}", RANDOM_ID)
    mt = mt.replace("{TIMEFRAME}", TIMEFRAME)
    mt = mt.replace("{COUNT}", COUNT)
    mt = mt.replace("{COMMAND_1}", COMMAND_1)
    mt = mt.replace("{COMMAND_2}", COMMAND_2)
    mt = mt.replace("{COMMAND_3}", COMMAND_3)

    st = st.replace("{RANDOM_ID}", RANDOM_ID)
    st = st.replace("{TIMEFRAME}", TIMEFRAME)
    st = st.replace("{COUNT}", COUNT)
    st = st.replace("{COMMAND_1}", COMMAND_1)
    st = st.replace("{COMMAND_2}", COMMAND_2)
    st = st.replace("{COMMAND_3}", COMMAND_3)

    with open(mec_path, "w") as out:
        out.write(mt)


    with open(sec_path, "w") as out:
        out.write(st)


eof_rule = """
#
# SEC Performance Test Rule
# Look for EOF at the end of the line, and send ourselves
# a USR1 signal to dump statistics, and a TERM signal to
#end the program.
type=Single
ptype=RegExp
pattern=EOF\s*$
desc=$0
action=eval %k ( $pid=$$$; kill (USR1, $pid); kill(TERM, $pid));
"""


with open(f"{SEC_OUTPUT_FOLDER}eof{SEC_SUFFIX}", "w") as out:
    out.write(eof_rule)
```

```
'''
SEC template:
# whoami
# $1 - hostname
# $2 - executable
# $3 - username
type=Single
ptype=RegExp
pattern=<\d+>\S+\s+\d+\s\d\d:\d\d:\d\d\s(\S+).*Process Create.*OriginalFileName
:\s+((?i){COMMAND_1}.exe).*User: (\S+)
desc=$0
action=event {RANDOM_ID}_for_$3_on_$1

# quser
type=Single
ptype=RegExp
pattern=<\d+>\S+\s+\d+\s\d\d:\d\d:\d\d\s(\S+).*Process Create.*OriginalFileName
:\s+((?i){COMMAND_2}.exe).*User: (\S+)
desc=$0
action=event {RANDOM_ID}_for_$3_on_$1

# hostname
type=Single
ptype=RegExp
pattern=<\d+>\S+\s+\d+\s\d\d:\d\d:\d\d\s(\S+).*Process Create.*OriginalFileName
:\s+((?i){COMMAND_3}.exe).*User: (\S+)
desc=$0
action=event {RANDOM_ID}_for_$3_on_$1

# collector
# $1 - username
# $2 - hostname
type=SingleWithThreshold
ptype=RegExp
pattern={RANDOM_ID}_for_(\S+)_on_(\S+)
desc=$0
action=write - {RANDOM_ID}: Quick execution of a series of suspicious commands
detected on host $2 from user $1
window={TIMEFRAME}
thresh={COUNT}

MEC Template:
title: Quick Execution of a Series of Suspicious Commands
id: {RANDOM_ID}
description: Detects multiple suspicious process in a limited timeframe
logsource:
    category: process_creation
    product: windows
detection:
    selection:
        CommandLine:
            - {COMMAND_1}
            - {COMMAND_2}
            - {COMMAND_3}
    timeframe: {TIMEFRAME}s
    condition: selection | count() by MachineName > {COUNT}
'''
```

Martin Ingesen

Real-Time Event Correlation for Windows Event Logs

NTNU
Kunnskap for en bedre verden