TU WIEN Informatics

Semantic approaches to detect file system log events for analyzing data exfiltration

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Agnes Fröschl, BSc
Matrikelnummer 1328403

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Priv.Doz. Mag. Dipl.-Ing. Dr. Edgar Weippl
Mitwirkung: Univ.Ass. Mag.rer.soc.oec. Dr. Elmar Kiesling
Dr. Andreas Ekelhart

Wien, 15. Oktober 2020

Agnes Fröschl	Edgar Weippl

TU WIEN Informatics

Semantic approaches to detect file system log events for analyzing data exfiltration

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering and Internet Computing

by

Agnes Fröschl, BSc
Registration Number 1328403

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.Doz. Mag. Dipl.-Ing. Dr. Edgar Weippl
Assistance: Univ.Ass. Mag.rer.soc.oec. Dr. Elmar Kiesling
Dr. Andreas Ekelhart

Vienna, 15th October, 2020

Agnes Fröschl | Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Agnes Fröschl, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. Oktober 2020

Agnes Fröschl

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt haben:

- Manfred und Max, die sich die Mühe gegeben haben diese Arbeit Korrektur zu lesen.
- Herrn Elmar Kiesling und Herrn Andreas Ekelhart, die meine Arbeit laufend begutachtet haben, mir hilfreichen Anregungen und konstruktive Kritik gegeben haben.
- Herrn Edgar Weippl für die Betreuung der Arbeit.
- Meine Familie, die mich immer wieder ermutigt hat diese Arbeit fertig zu stellen.

Kurzfassung

Daten sind heutzutage ein wichtiges Wirtschaftsgut für Unternehmen. Daher können data leaks zu schwerwiegenden Reputationsschäden führen und sich negativ auf den Umsatz des betroffenen Unternehmens, sowie dessen Kunden und Geschäftspartner auswirken. Diese Arbeit beschäftigt sich mit der Integration semantischer Technologien, um den Prozess der forensischen Analyse von Dateiaktivitäten zu unterstützen. Daher werden Dateisystemlogdaten semantisch dargestellt und über ein (nahezu) Echtzeitsystem analysiert. Das entwickelte Prototypensystem integriert *Logstash* und *TripleWave*, semantische Ontologien und *C-SPARQL*, um eine automatisierte Analyse der Dateizugriffsereignistypen bereitzustellen. Ein weiteres Ziel des Systems ist die Rekonstruktion von Dateilebenszyklen, die darauf abzielt, frühere Dateiaktivitäten zu verknüpfen, um verdächtige Muster wie Dateikopiervorgänge an externe Speicherorte zu identifizieren. Die Integration von Hintergrundwissen unterstützt einen Analysten beim Verständnis der Dateiaktivitäten und ihres Kontexts. Um das System zu bewerten, führen wir zunächst Leistungstests für Ereignisse mit einzelnen und gemischten Dateioperationen durch. In diesem Aufbau variieren wir auch die Parameter (z. B. die Zeit zwischen aufeinanderfolgenden Ereignissen), um Schwellenwerte und Einschränkungen zu identifizieren. Schließlich zeigen wir die Möglichkeit der Erstellung von Dateilebenszyklen in einem realistischeren Szenario mit mehreren Clients. In diesem Szenario wird auch die Verwendung von Hintergrundwissen (z. B. Benutzer- und Dateispeicherortkategorisierung) eingeführt, um erweiterte Abfragen und Ergebnisse zu ermöglichen. Während der Evaluierung stießen wir, aufgrund von Leistungseinschränkungen von *C-SPARQL*, auf Einschränkungen bei der Echtzeitanalyse von Dateisystemlogdaten. Reduktionen bei der Ereigniserkennung hängen von der ausgeführten Dateiaktivität und von der Rate der eingehenden Logeinträge ab. Darüber hinaus benötigen wir eine optimierte Fenstergröße von *C-SPARQL* Konstruktionsabfragen, um eine optimierte Balance zwischen der Häufigkeit erkannter Ereignisse, den akzeptablen Overhead und die Verzögerung der Benachrichtigungszeit zu erreichen. Darüber hinaus vergleichen wir konzeptionell unsere Ansätze mit bestehenden Open Source- und kommerziellen Lösungen, die ähnliche Ziele verfolgen.

Abstract

Data is an essential asset in today's organizations, and hence, data leakage can lead to severe reputation damage and negatively impact revenues of the affected companies, customers, and business partners. This thesis introduces an approach to integrate semantic technologies in order to assist the process of forensic analysis of file activities. To this end, file system log data is represented semantically and analyzed via a (near) real-time system. The developed prototype system integrates *Logstash* and *TripleWave*, ontologies, and *C-SPARQL* in order to provide an automated analysis of file access event types. A further goal of the system is the reconstruction of file life-cycles, which aims to link past file activities in order to identify suspicious patterns, such as file copy operations to external locations. The integration of background knowledge supports an analyst in understanding file activities and their context. To evaluate the system, we first conduct performance tests on single and mixed file events. In this setup, we also vary the parameters (e.g., the time between successive events) to identify thresholds and limitations. Finally, we demonstrate the possibility to construct file life-cycle graphs in a more realistic scenario with multiple clients. This scenario also introduces the use of background knowledge (e.g., users and file location categorization) to allow for enriched queries and results. During the evaluation, we encountered restrictions on a near real-time analysis of file system log data, due to performance limitations of *C-SPARQL*. Constraints on the event detection depend on the type of file activity performed and on the rate of incoming log entries. In addition, the window size of *C-SPARQL* construct queries has to be well balanced, in order to compensate the frequency of detected events, acceptable overhead, and delay in notification time. Furthermore, we compare our approaches with existing open source and commercial solution which follow similar goals.

Contents

CHAPTER 1

Introduction

In today' digital era, data has become one of the most important assets for organizations. Consequently, customer data as well as business information is sensitive, and hence, a company needs to protect these assets from unauthorized access, including e.g., criminals, competitors, and hacktivists [Cheng et al., 2017, Chismon et al., 2014].

Data leakage poses serious threats to organizations, potentially leading to severe reputation damage, a negative impact on trustworthiness and revenues of the affected companies, but also harms their customers and business partners. As the volume of data is growing exponentially and data breaches are happening more frequently than ever before, detecting and preventing data loss has become one of the most pressing security concerns for enterprises [Kurniawan et al., 2019a,b, Ullah et al., 2017, Torsteinbø, 2012]. Due to this situation, numerous countermeasures have been proposed [Ullah et al., 2017].

Digital forensics is a key method in identifying and analyzing data breaches. This thesis explores a semantic approach to assist digital forensic analysis processes. Ontologies help to integrate data across platforms. In addition, we provide the enrichment of existing knowledge and support graph-pattern based querying for forensic investigations.

In the following sections, we describe cases of past data breaches and highlight the importance of data security for today's enterprises. Furthermore, we include problems and challenges in the field of digital forensics and data loss prevention systems. Finally, we outline the aim of the work as well as our research questions.

1.1 Motivation

Data loss detection and prevention mechanisms become more and more important. Cheng et al. [2017] stated that data breaches influence the success of an enterprise. The loss of sensitive information can lead to significant reputation damage and financial losses of an organization [Cheng et al., 2017]. Therefore, the protection of data is one of

the most pressing security concerns for enterprises [Cheng et al., 2017]. Consequently, the organization's digital resources have become one of the most critical and sensitive components [Chismon et al., 2014]. These resources can be the intellectual properties of a company, including e.g., research and development achievements or processes desired by competitors [Chismon et al., 2014]. In addition, sensitive data also includes employee/customer data in terms of financial information, e.g. credit card numbers or account balances, and medical records [Cheng et al., 2017].

Websites such as *Techworld*[1] and *Information is Beautiful*[2] published articles of past data breaches from 2009 until 2019. These reports describe cases in which data leaks compromised sensitive data from big companies such as *Google*, *Facebook*, *FIFA*, *Quora*, *Box*, *Yahoo*, *Uber*, *Amazon*, *FedEx* and many more. Leaked data included customer data, email addresses, passwords, social security numbers and personal information.
Troy Hunt discovered the largest collection of leaked data at the beginning of 2019, which involved 772 million email addresses and 21 million passwords [Hunt, 2019]. Numerous individual data breaches collected the data from thousands of different sources. Hunt suspected that attackers intended to use the data for credential stuffing [Techworld, 2019]. Credential stuffing is a cyberattack in which a third party uses credentials obtained from a service's privacy breach to log in to another unrelated service.
Private user information was also compromised by vulnerabilities in the code of cloud storage services and social media websites. This was the case by *Box*'s sub-domain URL service [Herstein, 2019] ,*Facebook*'s "View As" tool and the APIs of the consumer version of *Google+* [Thacker, 2018]. Vulnerabilities enabled attackers to steal documents from personal *Box*' accounts, profile information from *Facebook* [Carrie, 2018] and data from the friends of users on *Google+*.
Invaders not only stole private information but also thieved credit card data in 2013, affecting *Target Corporation* [Cheng et al., 2017]. The thieves stole in this data breach 40 million credit card accounts. The incident has been called one of the most devastating data breaches in history [Shu et al., 2017].
The EU introduced the *General Data Protection Regulation* (GDPR) in 2018. Since then companies require to better safeguard personal data from cybercriminals. The first prominent case which violated these regulations was British Airways [BBC, 2019]. In this case, affected data includes login data, payment cards, and travel booking details, as well as name and address information. The company got a penalty of about 183 million pounds which exceeded the highest previously penalty by 367 times. This was due to the fact that Europe's GDPR allows fines of up to 4% of the annual turnover [BBC, 2019].

Data breaches or data loss can have different reasons. Moreover, the release of information can happen intentionally or accidentally. Intruders can perform data theft and inside attackers can cause data loss by sabotage [Cheng et al., 2017]. Due to the high utilization of modern communication channels, data leak vectors of internal data theft are increasing.

[1] https://www.techworld.com/security/uks-most-infamous-data-breaches-3604586, accesses: 10-02-2020

[2] https://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/, accesses: 10-02-2020

These vectors are cloud sharing, email, web-pages, instant messaging, social networks and so on [Cheng et al., 2017]. Thereby, employees can easily perform data exfiltration by using legitimate sharing platforms or communication channels [Cotenescu and Eftimie, 2017]. In addition, insiders usually know how to achieve the greatest impact whilst leaving little evidence [Colwill, 2010], which makes detecting internal data leakage incidents extremely challenging [Cheng et al., 2017]. Also, outsourcing can lead to the fragmentation of protection barriers and controls and increase the number of people treated as full-time employees [Colwill, 2010]. According to the *CyberSecurity Watch Survey*, conducted by the U.S. Secret Service, damage caused by insider attacks was more severe than damage from outsider attacks [Cotenescu and Eftimie, 2017].

In spite of access control regulations and investments made on security control measures and other security-related products, actions from internal employees, like sharing data or transmitting confidential data intentionally or accidentally, can cause serious risks and can have major negative impacts [AlHogail, 2017] [Colwill, 2010].

1.2 Problem Statement

A major problem of data security in enterprises is the lack of visibility on who created which data, where it resides, and who has access to it. In order to protect data and prevent data exfiltration a variety of technologies exist. Countermeasures of data security focus on preventing data leaks, mitigating threats, and analyzing log data of past cases. Technologies include security policies, which are one of the most fundamental actions for threat mitigation [Awais Rashid et al., 2014] and are strongly used in so-called Data Loss Prevention (DLP) systems [Torsteinbø, 2012]. In addition, companies often use logging and monitoring systems to mitigate data security issues [Carrier, 2005]. A *Security Information and Event Management* (SIEM) system provides a solution for log management, aggregation and event correlation [Kostrecová and Bínová, 2015].
In case a prevention system becomes inefficient, the collection of log data helps for later analysis [Torsteinbø, 2012]. In order to retrieve digital evidence from log data, digital forensics acts as a supporting tool to identify and reconstruct events [Sindhu and Meshram, 2012]. Thereby, computer forensics plays an important role in the field of data security and aims to collect and document cyber attacks [Tripathi and Meshram, 2012, Torsteinbø, 2012].
Despite of technologies to prevent data leaks and tools to mitigate threats, enterprises cannot protect themselves from data leaks completely. Likewise, the extensive forensic analysis does not ensure successful detection of suspicious events.

Security policies can be inefficient due to technology and configuration weaknesses. This can involve different interpretations of the specification, which invaders then exploit. Also, weaknesses in policies can cause malicious traffic to bypass [Kotenko and Chechulin, 2012]. In addition, the analysis of log data collected by logging and monitoring applications holds its challenges. Limitations of a SIEM system range from misconfigurations, high costs involved, and time-consuming analysis due to the high volume of log data. Reports

of SIEM systems contain a lot of noise and often collected data which is not relevant for the analysis at hand. In order to find necessary information, enterprises often require adaptations [Kotenko and Chechulin, 2012]. The amount of log data collected for monitoring purposes can become overwhelming, which leads to challenges regarding good filters and alerting rules [Carrier, 2005]. An insufficient amount of alerting rules puts an enterprise at risk to miss potential threats. Otherwise, too many rules can possibly trigger an overwhelming number of false positives, which requires time to review alerts and a dedicated team. A challenge digital forensics faces is heterogeneous log formats which needs to be overcome first [Sindhu and Meshram, 2012]. Also, challenges are the large volumes of data and the time to acquire and analyze forensic media [Fahdi et al., 2013]. For the analysis process digital forensic usually uses a computer forensic laboratory, which needs much equipment in order to process forensic data and to perform examinations. Many analysts need to seize investigated media in order to prevent changes [Quintiliano et al., 2013]. Furthermore, a forensic analysis requires tools supporting the detection of suspicious events saved in log data. In regards to file system activities, a variety of open source and commercial tools exist. However, challenges and problems of these tools are restrictions to log data of only one operating system, and high complexity that requires technical experienced employees for their use. We provide a survey of existing tools in Chapter 4.

Generally, weaknesses of existing exfiltration detection tools are the high complexity of the system itself and the company expenses needed. Configuring security policies and logging management systems are very time consuming tasks. In addition, a forensic analysis requires adequate equipment and a time consuming analysis. Thereby, additional problems can arise which require a dedicated team of experts in the field of data security. Torsteinbø [2012] mentions that the amount of investment and costs associated can motivate the management to consider a different approach to existing prevention systems.

1.3 Aim of the Work

In this work, we focus on an analysis of file system events via a semantic approach in near real-time. Thereby we attempt to trace all file creations, accesses, and modifications as well as any sharing of files between clients.

This thesis addresses the following questions:

1. How and to which extend can we use semantic approaches to help the analysis process of files system log data?

2. Can we use information, gathered by a semantic representation of file system events and by the construction of the file life-cycle, to detect potential file exfiltration?

We define the requirements for our system in Section 2.1. In Section 8.1 we revisit our research questions.

1.4 Structure of the Thesis

We structure the thesis as follows:

- In Chapter 2 we describe how we apply the principles of design science as our methodological approach.
- In Chapter 3 we provide an overview of the state of the art of current approaches considering data exfiltration attack vectors, implications and existing countermeasures. In addition, we describe background knowledge concerning Semantic Web technologies, file system events and existing languages used for semantic complex event processing.
- In Chapter 4 we describe existing approaches concerning the monitoring of the file system and computer forensic techniques to analyze the file system. In addition, we present existing solutions of semantic approaches in the field of forensic analysis and give an overview of the current state of semantic representations of log data.
- In Chapter 5 we describe the requirements for our architecture and models in order to present file system events semantically. In addition, we also present a vocabulary for background knowledge and describe the concept for reconstructing file life-cycles.
- In Chapter 6 we describe the implementation of our prototype system including external tools, preconditions, adaptions and components.
- In Chapter 7 we present test scenarios and evaluate their results concerning our defined research questions.
- In Chapter 8 we conclude this thesis. We describe open issues and limitations of our implementation. Furthermore, the chapter outlines directions for future work.

CHAPTER 2

Methodological Approach

The following sections describe how this thesis uses the principles of design science as a methodological approach. This concept acts as a guideline to create and evaluate new artifacts in the field of Information Systems (IS). The principles of the methodology incorporate the environment in which we develop our research results. When describing the environment we define requirements and create the foundation for later testing of artifacts. It gives instructions to define a knowledge base and incorporates existing solutions and methods when setting up the design of artifacts. Furthermore, the continuous evaluation during the design process of artifacts ensures that requirements are met.

Figure 2.1 shows the three cycles in design science, which include the relevance cycle, the design cycle, and the rigor cycle. According to [Hevner, 2007] the relevance cycle bridges the contextual environment of the research project with the design science activities. The central design cycle iterates between the core activities to construct and evaluate artifacts. The third cycle, called rigor cycle, connects the design science activities with the knowledge base of scientific foundations, experience, and expertise that informs the research project.

Figure 2.2 illustrates our version of the three cycles. The application domain includes forensic analysts and a network of clients that run on macOS. Defined problems are data exfiltrations of sensitive files. Our opportunities consist of a near real-time approach to detect suspicious file activities. Our knowledge base describes methods and existing tools which assist the construction of artifacts. This includes Semantic Web technologies, external tools, documentation, and solutions with similar approaches. We incorporate external tools for collecting and processing log data, and transforming data into an RDF data stream. The artifacts of the design cycle consist of defined ontologies, implemented processes for handling log data, methods to link independent file events, and the reconstruction of the history of file activities. We run performance tests and simulate a real-world scenario in order to evaluate our artifacts. Section 2.3 describes the evaluation and the artifacts in more detail.

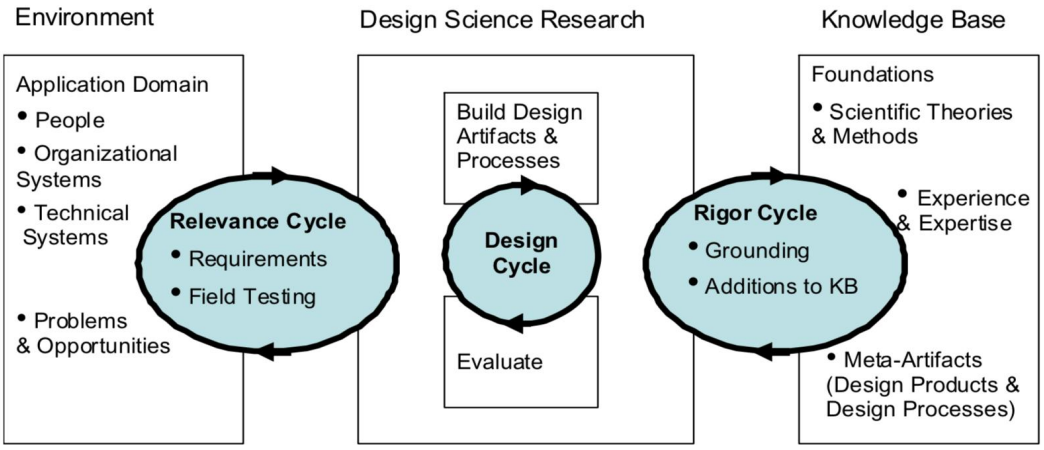


Figure 2.1: Three cycles of design science research [Hevner, 2007]

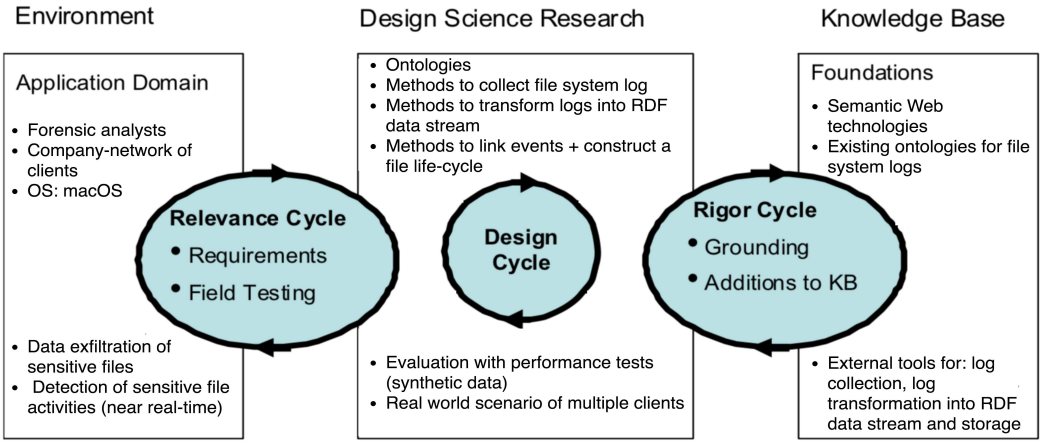


Figure 2.2: Our instances of all three cycles of design science

Before we can develop any artifacts, we require an iteration of the relevance cycle and the rigor cycle. The output of both cycles acts as input and initialization of the design cycle. The three cycles are heavily based on the conceptual framework for understanding, executing, and evaluating *Information System* (IS) research, presented by Hevner et al. [2004].

In the following sections, we first explain the importance of the relevance cycle and the rigor cycle and how we apply their practices in this thesis. Furthermore, we give an overview of the steps we implement in the design cycle.

2.1 Relevance Cycle - Application Domain

The relevance cycle defines requirements and the application domain, consisting of the people, organizational systems, and technical systems that interact to work toward a goal. In addition, the first cycle also defines acceptance criteria for the evaluation of the research results [Hevner, 2007].

Considering the scope of this thesis — monitoring file activity to detect/investigate data exfiltration — the overall application context is an organizational environment that uses the resulting artifacts of the research. Within an organizational network, we focus on file activities, such as monitored directories within the operating system, USB directories, and folders of cloud storage providers such as *Dropbox*.

The people and roles in (passive) contact with the developed artifacts (monitoring system) include those users that work with the monitored data within the organizational boundaries. The users who directly interact with the resulting artifacts (analysis system) have different roles within a cyber-security or information security team. Their responsibility is to mitigate risks and to perform forensic analysis on logged data as a response to data exfiltration threats or incidents.

We have outlined the problems and requirements of the artifacts in Section 1.2. These are the result of a literature study in which we identified security issues with an emphasis on data exfiltration. Following the aim of this work, we intend to solve problems created by challenges of forensic analysis activities as well as to avoid data leaks of sensitive organizational data.

We describe our requirements in the following listings:

1. Log data (in particular file system logs) should be processed by a near real-time streaming approach.
2. An ontology for file system log data should be developed.
3. Collected log data should be analyzed automatically.
4. All activities of a file should be traced in order to reconstruct a file's life-cycle.

In summary, this thesis aims to create a solution for analyzing log data in order to recognize patterns of potential data exfiltration to protect sensitive data within an organization. Furthermore, we aim to investigate if the system can increase the time needed for data analysis activities. Therefore, we aim to apply a near real-time streaming approach.

2.2 Rigor Cycle - Foundations

The knowledge defined in the rigor cycle should act as support for building artifacts and to build the scientific foundation of the design science research [Hevner, 2007].

In this thesis, we leverage existing semantic methods, including ontologies, Semantic Web technologies, and tools which help to extract file system log data and represent its content semantically. Furthermore, we build on existing solutions for parsing log data and querying semantic data. In addition, approaches and experiences in the field of forensic analysis advise this thesis. This includes studying existing approaches for analyzing file system activities, and semantic solutions in the field of forensic analysis. Chapter 4 describes the current state of the art in detail.

Prior to developing approaches, we thoroughly study existing solutions and directions. We found literature that describes generally used technology. Chapter 3 describes the backbone of our study.

2.3 Design Cycle Iterations

After defining the relevance and rigor cycle, we can initialize the design cycle, which represents the heart of the research. We iterate between creating and evaluating each artifact until the requirements are met, which we derive from the relevance cycle. These iterations provide us with a continuous feedback in order to improve the design [Hevner, 2007].

Generally, our artifacts are ontologies, models, and methods (e.g. *Semantic Web* methods) applied in the development of this thesis. These artifacts support gaining new knowledge and the problem domain becomes easier to understand [Vaishnavi et al., 2004] [Hevner, 2007].

The following listing summarizes the artifacts we aim to achieve in sequence:

1. An ontology for representing file system log data.
2. Ontologies for describing background knowledge for file activities.
3. Determine methods and practices which help to collect, parse, and store log data semantically.
4. Construct methods to analyze file activity (e.g., to recreate the life-cycle of specific files).

We start our research by defining an ontology to represent file system log data semantically. Therefore, we conduct several iterations of the design cycle in order to develop a vocabulary that fits file system logs for different operating systems. The study of existing ontologies in this domain helps us to build our representation. In each iteration, testing ensures if the system covers all information by the currently used ontology. Thereby, we test our vocabulary against test cases in order to verify that requirements are met. Chapter 7 describes our evaluated test scenarios. After we finish the required log format ontologies, we continue to define a background knowledge vocabulary. Subsequently, we need to define

an architecture for processing, saving, and analyzing log data automatically. Several cycles refine the built prototype system until defined requirements are met accordingly. We define our requirements in Section 2.1.

We perform several scenarios to evaluate if our research artifacts meet the requirements and in order to answer the defined research questions from Section 1.3. Section 7.1.3 explains the evaluation setup in more detail. Section 7.1.2 describes scenarios used for the analysis. Used scenarios consist of test runs with synthetic log data and a real-world scenario including several clients. Runs with synthetic log data represent performance tests of our system. Thereby, we aim to provide different ranges of waiting times between events and sequences of file activities. The aim of this iterations is to find the limitations of our system.

The real-world scenario aims to evaluate the use of defined background knowledge in order to detect file activity patterns leading to data exfiltration. We aim to retrace which person performed specific operations and which data channels the user used in order to transfer files. In addition, we use data from this scenario to evaluate the reconstruction of the file life-cycle. Section 7.2 describes the results and statistics of the defined scenarios in more detail.

In summary, this cycle consists of definitions, implementations, and evaluation activities in order to create an ontology for file system log data, automated parsing of logs as well as analyzing and correlating activities of parsed data.

CHAPTER 3

Background

The following sections contain background information on the underlying areas of this thesis. We collected the information as part of the rigor cycle, which builds the foundation of our research. We describe the research methodology in Chapter 2.

We start with a broad discussion on data exfiltration. Section 3.1 gives an overview of possible attack vectors and motivations for data theft. Furthermore, it includes implications caused by attacks and countermeasures used to prevent data exfiltration. Section 3.2 gives an overview of used *Semantic Web Technologies* and defines important terms in this area. Section 3.3 describes the information of file system events which we represent as RDF data in this thesis for analysis. Section 3.4 discusses implementations of Semantic Complex Event Processing engines which aim to handle and analyze complex relations over RDF data.

3.1 Data exfiltration

Data exfiltration is the unauthorized transfer of data from a computer. The term is also referred to as data extrusion, data exportation, data leakage, or data theft. Data exfiltration of sensitive data is one of the main targets of cyber-attacks [Ullah et al., 2017]. Insiders or outsiders of an organization can perform these attacks.

Customer data is leaked often, followed by confidential information, health records, and lastly intellectual property. Gordon [2007] discusses in his work that 52% of data security breaches stem from internal sources compared to the remaining 48% caused by external hackers. He further breaks down the number of internal data leaks caused by malicious intent (1%) and inadvertent data breaches (96%).
Cheng et al. [2017] conducted a study that showed that internal employees account for 43% of corporate data leakage, and half of these leaks are accidental. Corporate espionage, financial reward, or a grievance with their employer motivated attackers [Gordon, 2007,

Cheng et al., 2017].
Implications caused by data leakage of confidential data, such as customer data, can cause legal consequences. Data leakage poses serious threats to organizations, including significant reputational damage and financial losses [Cheng et al., 2017]. Data exfiltration causes considerable costs, including fines from regulatory bodies and the cost of IP theft, which may lead to significant economic losses. Furthermore, a decrease in a business's reputation and sales can be a result of stolen sensitive data [Awais Rashid et al., 2014].

Various attack vectors cause data leakage. In order to prevent data exfiltration, a large number of countermeasures exist, which aim to detect, prevent, and investigate theft of sensitive or private data. Awais Rashid et al. [2014] describes security policies as the most fundamental measures for mitigating exfiltration threats. This method of controlling the access to sensitive files is strongly used in so-called Data Loss Prevention (DLP) systems [Torsteinbø, 2012]. A DLP system is a strategy in order to prevent end-users to send sensitive or critical information outside the corporate network. Policies can range from policies that grant and maintain access to files and storage of sensitive material [Awais Rashid et al., 2014]. Policies can also involve blocking the access of specific web sites [Gordon, 2007] or encrypting data before it is sent to a cloud storage device [Awais Rashid et al., 2014]. Other policies might aim to completely block any SSL traffic in order to prevent any *SSL Tunnelling* tactics that users are utilizing to bypass security measures such as permitter firewalls and antivirus software.

In addition to security policies, Ullah et al. [2017] classifies data exfiltration attack vectors used by external attackers and potential countermeasures. Examples of these countermeasures are proactive activities to resist against data exfiltration attempts. The endpoint devices (such as PCs, Laptops, and Servers) incorporate these countermeasures to control access to the data resided on these devices or apply particular security tactics (such as encryption, data classification, and cyber deception) to help secure data against exfiltration attacks. Other countermeasures aim to detect exfiltration attempts. The network-level or host-level deploys these to monitor the network traffic and host access patterns to either look for transfer of sensitive information or observe abnormalities. Ullah et al. [2017] also presents countermeasures which investigate data exfiltration incidents such as forensic analysis in order to fix security weaknesses and taking appropriate actions after a breach. In addition, Gordon [2007] provides another mitigation tactic against data theft, which is the implementation of a *Secure Content Management Solution*. This solution includes techniques such as lexical analysis of traffic passing through a specific device on the network and fingerprinting. Based on patterns and keywords in passing messages, the system categorizes the traffic acts on it accordingly, which can pass, quarantine, notify, or block the message. This tactic should mitigate the threat of releasing confidential information through electronic channels, which includes email, FTP, HTTP, Webmail, Instant Messaging, and removable storage devices. Awais Rashid et al. [2014] defines the use of outbound FTP or HTTP/HTTPS connections as the most common data exfiltration strategies as more than 50% of analyzed data breach incidents favors these exfiltration modes. It blends in with normal network traffic and is hard to

distinguish from legitimate activities of users.

3.2 Semantic Web Technologies

The term "Semantic Web" is closely related to the World Wide Web and was coined by its inventor, Sir Tim Berners-Lee [Hitzler et al., 2010]. The Semantic Web aims to build abstract models to describe the world, which enables an easier understanding of a complex reality in order to allow machines to automatically come to reasonable conclusions from encoded knowledge [Hitzler et al., 2010]. Furthermore, it is a tool to exchange information that allows us to distribute, interlink, and reconcile knowledge on a global scale [Hitzler et al., 2010].

In this section, we give an overview of Semantic Web technologies, which provide the foundation for this thesis. The most important Semantic Web technologies, concepts, and methods relevant to this thesis are OWL/RDFS, RDF, SPARQL, and C-SPARQL. The following segments give a short overview and definition of these technologies.

OWL/RDFS The Web Ontology Language (OWL)[1] is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things. OWL includes the Resource Description Framework Schema (RDFS). Therefore, RDFS is less expressive than OWL. Both languages are used to define ontologies.

Ontology In computer science, an ontology is a description of knowledge about a domain of interest, the core of which is a machine-processable specification with a formally defined meaning [Hitzler et al., 2010].

RDF The Resource Description Framework (RDF) is a formal language for describing structured information. The goal of RDF is to enable applications to exchange data on the Web while still preserving their original meaning [Hitzler et al., 2010].

SPARQL SPARQL[2] is an RDF query language for querying and manipulating RDF graph content on the Web or in an RDF store.

C-SPARQL Continuous SPARQL (C-SPARQL)[3] is a language for continuous queries over streams of RDF data. C-SPARQL queries consider windows that allow us to observe the most recent triples of such streams, while data is continuously flowing. C-SPARQL extends the semantic query language SPARQL and is an example of an RDF Stream Processing (RSP) system, which combines the advantages of semantic web technologies and traditional data flow management systems [Dia et al., 2017].

[1] `https://www.w3.org/OWL/`, accessed: 03-06-2019
[2] `https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/`, accessed: 23-04-2019
[3] `http://streamreasoning.github.io/TripleWave/docs.html`, accessed: 03-06-2019

3.3 File System Events

Semantic Web technologies allow computers to combine and process collected data based on the meaning that this content has to humans [Hitzler et al., 2010]. In this section, we define file system events, which we aim to transform into RDF data. We use the transformed data to analyze patterns of file activities.

File system logs contain information about past file activities. These logs build the backbone for capturing *File Access Events*, which define information about performed user interaction with the corresponding file. In order to detect the actual event, we start at file operation types and split it into its low-level system calls. These low-level calls are the underlying access calls of each file operation contained in file system log data. We represent these file operation types as:

a) move **b)** create **c)** edit **d)** rename **e)** copy and **f)** delete file operations.

Depending on the *File Access Events*, we require to correlate several file system log entries to construct the event. However, in some cases, only one log entry contains all the information that we require in order to define its higher *File Access Events*.

As an example, the file operation *rename* triggers a single log entry that contains all the information needed. However, a *copy* operation involves several user interactions. First, actions include the access of a file to copy. In the next step, the operation copies the file to the clipboard and lastly paste the file into a new location. Consequently, the file-system produces several events. We require the identification of all involved events in order to gain knowledge about a performed copy operation.

Information concerning file-system events is mainly required for defining our semantic model which we describe in Section 5.3.

3.4 Semantic Complex Event Processing (SCEP)

In the following sections, we provide definitions about important terms in the context of SCEP and aim to describe existing languages that can be used in order to process RDF data automatically.

3.4.1 Definition of terms

Complex Event Processing (CEP) is an important real-time computing paradigm for analyzing continuous data streams [Zhou et al., 2016]. Semantic Complex Event Processing integrates Semantic Web technologies into CEP. Each event is essentially a set of RDF triples: resource, property, and value [Schaaf et al.].

The combination of Semantic Web technologies and CEP can be used to enrich event data and metadata with the domain knowledge from ontologies [Schaaf et al.]. Furthermore,

static background information (static RDF datasets or ontologies) can be employed to reason upon the context of detected events [Gillani et al., 2017].

Gillani et al. [2017] discusses that most CEP systems consider a relational data model for streams and their proposed languages and optimizations are also tightly coupled with the model. The paper introduces a semantic approach to overcome issues of integrating and analyzing data coming from different data stream sources, with varying formats. Existing solutions offer the use of RDF as a unified data model for integrating diverse data sources across heterogeneous domains. Also Zhou et al. [2016] mentions that existing work on CEP is largely limited to relational query processing. According to the paper, a semantic approach on CEP allow accessible analytics over data streams that have properties from different disciplines and help to process this analysis in real-time over a high volume of data.

3.4.2 SCEP Languages

At the current state, a standard query language for expressing continuous queries over RDF graph streams does not exist. The RSP Group is still working on the definition of a common model for querying RDF data streams [Keskisärkkä, 2017]. Important query language include C-SPARQL [Barbieri et al., 2010a], CQELS [Le-Phuoc et al., 2011]), $\text{SPARQL}_{\text{stream}}$ [Calbimonte et al., 2010]), Sparkwave [Komazec et al., 2012], EP-SPARQL [Anicic et al., 2011], INSTANS [Rinne and Nuutila, 2016], SPAseq [Gillani et al., 2017] and C-Sprite [Bonte et al., 2019]. These languages represent an RSP implementation and aim to combine the principles of the Semantic Web with stream processing and CEP [Keskisärkkä, 2017].

C-SPARQL [Barbieri et al., 2010a] is a query language that extends SPARQL 1.1 to support the processing of RDF triple streams. The language supports both time-based and count-based windows over the most recent portions of a stream. C-SPARQL queries can combine triples from more than one RDF stream.

CQELS (Continuous Query Evaluation over Linked Streams) [Le-Phuoc et al., 2011] is a native and adaptive query processor for unified query processing over Linked Stream Data and Linked Data. In contrast to the existing systems, CQELS uses a "white box" approach. CQELS provides a flexible query execution framework with the query processor dynamically adapting to the changes in the input data. The user is able to define multiple windows over the same stream within a single query, a feature no other RSP implementation supports [Keskisärkkä, 2017].

$\text{SPARQL}_{\text{stream}}$ [Calbimonte et al., 2010] is a language that is similar to C-SPARQL. The main difference is that $\text{SPARQL}_{\text{stream}}$ only supports time-based windows. An upper and lower time bound defines those windows [Calbimonte et al., 2010]. Sparkwave [Komazec et al., 2012] is an RSP implementation that supports continuous reasoning over RDF data streams and time-based windows. However, Sparkwave has some known limitations concerning the size of the supported background knowledge and the amount of reasoning functionalities [Komazec et al., 2012].

EP-SPARQL [Anicic et al., 2011] is a SPARQL 1.0 extension designed for event processing, and it explicitly supports temporal operators as part of the language. EP-SPARQL supports RDF triple streams but assumes that a time interval representing a valid time annotates all triples. The execution engine for EP-SPARQL translates queries into the ETALIS Language for Events (ELE), which the ETALIS engine also executes. The language provides functions to access start time, end time, and duration of a matched graph pattern but does not support to define windows over streams [Keskisärkkä, 2017].

INSTANS (Incremental eNgine for STANding Sparql) [Rinne and Nuutila, 2016] is a platform for executing continuous queries using standard SPARQL and SPARQL Update. The focus of INSTANS is to support CEP and incremental processing of queries. INSTANS does not support windows over streams. However, the same results can be computed by indirect mechanism [Rinne and Nuutila, 2016].

SPAseq [Gillani et al., 2017] is an implementation of a SCEP query language. The language extends SPARQL with new Semantic Complex Event Processing (SCEP) SCEP operators that the user can evaluate over RDF graph-based events. SPAseq supports the expression of temporal operators, conjunction, disjunction, and event selection strategies. It also provides an option to use multiple heterogeneous streams. SPAseq uses a non-deterministic automata (NFA) model for the evaluation of the SPAseq queries [Gillani et al., 2017].

C-Sprite [Bonte et al., 2019] represents an optimized engine that should be at least twice as fast as current approaches. The authors claim that the algorithm operates in constant time and scales linearly in the number of continuous queries.

Even though numerous query languages for Semantic Complex Event Processing do exist, none of the defined languages supports exactly the same features. [Keskisärkkä, 2017] discusses most of the languages above and represents a listing of all features with an additional comparison of the implementations.

CHAPTER 4

State of the Art

This chapter describes the literature review of the underlying areas this thesis deals with. In the first section, we discuss solutions that monitor the host system to detect data exfiltration. Furthermore, we describe three provenance system, similar to our approach. We explain important aspects of computer forensic file-system analysis. In addition, we discuss commercial and open-source tools that assist in the process of forensic analysis. We present semantic approaches in forensic analysis and describe existing solutions of Semantic Web technologies in order to assist the computer forensic process. Furthermore, we examine the current state of presenting log data semantically and describe existing approaches in this manner.

4.1 Data exfiltration

Since we aim to monitor file-system access patterns, we are mainly interested in solutions that analyze the host system and provide methods to investigate suspicious file access patterns. Therefore, we will not discuss countermeasures such as security policies, which aim at restricting access in this thesis.

In contrast to the previously described solutions in Section 3.1, Awais Rashid et al. [2014] describes a host-based access analysis which is an alternative to monitoring network traffic. This approach monitors the storage system saving the data and detects unusual patterns of access [Awais Rashid et al., 2014]. Servers or employee machines use the solution as part of a database management system for a central share, or even at the file system or system call level. Simple logging and analysis of particular patterns of access to the file system, database, or the OS's system call library can trigger alerts for exfiltration.

Similarly to the previous approach, Krishnan et al. [2012] suggests using a *hypervisor*, for monitoring hosts, with a forensic audit log that records disk accesses, system calls to

the host, and chains of access to memory across processes.

This approach is comparable to detection systems described by Awais Rashid et al. [2014]. These systems can include email filters, which refuse to forward suspicious attachments, database management systems that refuse to respond to suspicious queries or gateways that refuse to forward packets containing sensitive data.

Two articles by Grier [2011] and Patel and Singh [2013] examine methods for detecting if a user copies a file. Copy operations usually leave no trace within file systems. Consequently, investigation tactics based on the file access timestamp do not distinguish copy operations from other forms of file access operations.
Grier [2011] presents a method to examine a file-system. The method determines if and when a user copies a file from the file-system. The author developed this method by stochastically modeling file-system behaviors. The method defines patterns based on the Media Access Control (MAC) timestamp of the operations to identify file copy actions. The detection of these patterns is difficult to distinguish from routine activities within the file-system and is quite similar to other access operations. However, according to the author this approach has not been tested in a real-world file system to determine its accuracy. Patel and Singh [2013] aims to extend this approach and proposes a technique using a fuzzy inference system in order to distinguish between legitimate and illegitimate copy operations.

4.2 Provenance Systems

The interest in provenance systems has grown in the scientific field of the Semantic Web in the last years [Pérez et al., 2018]. The term *provenance* is related to the word *lineage* and defines the entire amount of information regarding a piece of data This includes the source of the data, contextual information, dependencies, relations, and processing steps which lead to the current state of the data. Techniques of provenance systems allow the user to verify data products. Thereby, the user can determine its authorship and infer its quality. It provides the means to interpret and understand it. Also, it enables users to analyze the process of steps which lead to the result of the data product [Pérez et al., 2018].

The provenance system can assist the detection of file system activities in regards to who created a specific file, who modified it and when, and who moved or copied the file. Bates and Butler [2014] describe the system as a solution to gather and report metadata that describes the history of each object being processed. Thereby, a user can track and understand how a piece of data came to exist in its current state on the system. Furthermore, Ma et al. [2017] introduce provenance tracking as a critical component for cyber-attack investigations. The work describes the need for provenance tracking in order to understand the attack including its root cause and consequences. Existing provenance systems include e.g., the Linux Audit System [Shortridge, 2020], ProTracer [Ma et al., 2016] and the LPM enabled HiFi system [Bates and Butler, 2014].

Linux Audit System is used to track system activities and is a native feature of the Linux kernel [Shortridge, 2020]. The underlying daemon of the audit system is *auditd* [1], which logs system calls, file access activities, and pre-configured auditable events within the kernel. We use an adapted version of the audit daemon for macOS which is called *OpenBSM* in order to log any file activities. We define configuration steps and the usage of the service in Section 6.4.

ProTracer is an important approach in *Advanced Persistent Threat* (APT) attack detection [Ma et al., 2016]. The system is a lightweight provenance tracing system that alternates between two activities. Those activities include *audit logging* and *provenance propagation* (or *tainting*). The goal of the system is to support *what*-provenance and *how*-provenance queries on system objects, such as processes and files. A *what*-provenance query would search for the source of a process (or file) x, or investigate which other files were derived from x. An example query for *how*-provenance would search for activities that led to the corruption of file x. The results of a *how*-provenance query can be built as a causal graph. This approach can be compared to our solution of reconstructing a file history graph illustrating past file activities. Our file history graph describes any file operations leading to the current state of the monitored file. We describe our concept of reconstructing a file history in Section 5.7. In addition we directly link to background information such as the related user account and information to the exfiltration channel. Furthermore, we are able to query for events together with defined background knowledge.

LPM enabled HiFi system was designed by Bates and Butler [2014] and describes a *Linux Provenance Monitor framework* (LPM). The framework aims for the development of an automated, whole-system provenance collection system on the Linux operating system. The system includes a re-implementation of the *Hi-Fi system* presented by [Pohly et al., 2012]. Hi-Fi is a kernel-level provenance system that leverages the *Linux Security Modules* framework to collect high-fidelity whole-system provenance and thereby aims to collect malicious behavior within a compromised system [Pohly et al., 2012]. Bates and Butler [2014] designed LPM in such a way to enable experimentation with new provenance collection mechanisms and to support interoperability with other security mechanisms.

4.3 Forensic Analysis of a File System

Computer forensics plays an important role in the field of data security and assist in the analysis of cyber-attacks from various sources, which can affect computers, software, a network, an industry, or the Internet itself [Sindhu and Meshram, 2012].

Digital forensics is the science of identifying, extracting, analyzing, and presenting the digital evidence that has been stored in digital devices. Forensic tools and techniques are an integral part of criminal investigations used for inspecting suspect systems, gathering

[1] https://man7.org/linux/man-pages/man8/auditd.8.html, accessed: 2020-09-12

and preserving evidence, reconstructing events, and assessing the current state of an event [Tripathi and Meshram, 2012]. An analysis of file-system events is mostly done in the field of computer forensics. The aim of digital forensics is to find footprints left by a digital device in order to find evidence for a wide range of inquiries [Carrier, 2005].

In the next two sections, we focus on sources used for analyzing file-system events and prominent tools that assist in collecting digital evidence that we found during our literature review.

4.3.1 Sources for digital evidence within a file-system

The operating system is often the main source for digital evidence of file-system log data, in order to obtain information needed for a forensic analysis [Adelstein, 2006]. Lokhande and Meshram [2015] calls this type of forensic investigation *OS Forensic* and classify it as part of system forensics. OS Forensic collects information through Windows Registry, Event Viewer Log, and does kernel forensics.

Opsitnick et al. discusses various other information sources besides the operating system. These sources include logs of USB activities, logs of cloud storage providers, and information on if a user sends a file to a personal email account. A forensic analysis uses the information for forensic analysis in case of data theft by internal employees. Facts about USB activities and the usage on installed cloud storage providers, such as Dropbox, Google Drive, or Microsoft One-Drive, reveal information on several key facts. These facts include connections to a computer of the USB device and the timestamp of the connections and information if a user opened or accessed a file on a cloud storage provider [Opsitnick et al.].

4.3.2 Tools for File System Analysis

Various commercial and open-source forensic tools are already available which assist in collecting digital evidence. However, considering the aim of this thesis, which follows the approach on correlating file activities and analyzing the history of a file, the following tools are following a similar approach to our work:

Open-source tools

- Plaso[2]
- Plaso command line tools[3]: Log2timeline, pinfo, preg, psort
- Timesketch[4]

Commercial tools

[2] `https://github.com/log2timeline/plaso`, accessed: 2019-03-02
[3] `https://plaso.readthedocs.io`, accessed: 2019-06-03
[4] `https://github.com/google/timesketch`, accessed: 2019-03-02

- "Next-Gen Data Loss Protection" by *Code42*[5]
- ADAudit Plus[6]
- "Log&Event Manager" from *SolarWinds*[7]

Plaso aims to collect all timestamped events of interest on a computer system. The tool provides a computer forensic analysis, and the possibility to aggregate all events. Therefore, it helps to collect digital evidence from a storage media image or device, and present timelines of file system events. The system is a Python-based back-end engine and provides packages for macOS, Windows, and Linux. Moreover, Plaso already supports different log formats and users can create extensions of the tool by creating custom-defined parsers.

Log2timeline is a command-line tool designed to extract timestamps from various files found on a typical computer system and aggregates found events. The user can create Plaso files with the tool. Other tools such as *pinfo*, *preg* and *psort* analyze the Plaso storage file and provide information on the content. Moreover, users can also use the tools to analyze Windows Registry files and to post-process Plaso storage files.

Timesketch offers functionalities to present collected timelines by Plaso as graphs. The tool offers features for collaborative forensic timeline analysis. It aims to support forensic investigations by a full-text search, organize investigated events by adding labels and comments as well as share findings using saved views. Timesketch visualizes relationships between events as a graph [Berggren, 2017] which the tool provides next to a tabular view of investigated log data. Timesketch uses *Neo4j* as a graph database back-end and the query language called *Cypher*.

"Next-Gen Data Loss Protection" by Code42 offers a commercial platform that protects endpoint and cloud data from loss, leak, misuse, and theft. It offers visibility to all files and maintains a comprehensive history of every version of every file. Furthermore, the tool monitors file movements within cloud storage providers.

"ADAudit Plus" by ManageEngine offers features to track a Windows file server. The tool tracks file access, changes to documents in their files and folder structure, shares, and permissions. ADAudit Plus advertises to offer forensics of all file changes, failed attempts to file creations, deletions, modifications, and folder structures.

[5] `https://www.code42.com/product/`, accessed: 2019-03-02

[6] `https://www.manageengine.com/products/active-directory-audit/windows-file-server-auditing.html`, accessed: 2019-03-02

[7] `https://www.solarwinds.com/log-event-manager-software`, accessed: 2019-03-02

The "Log&Event Manager" from SolarWinds offers file tracking features and directory access monitoring of movements and shares. The tool acts as a file integrity monitoring tool to detect and alert on changes to key files, folders, and registry settings. The software correlates system, Active Directory, and file audit events to obtain information on which user is responsible for accessing and changing a file. System administrators can use this information to create an alert or run reports to review activities.

In summary, the open-source solutions Plaso combined with its command-line tools and Timesketch present a comprehensive array of commands and tools for timeline analysis. However, functionality and commands are often not documented, the tools are not self-explanatory, and the documentation is lacking in information and examples.
Downsides to the presented commercials tools are the involved costs and the restriction to only Windows file system events, which is the case for ADAudit Plus by ManageEngine.

We compare existing approaches to our solution in more detail in Section 7.3.

4.4 Semantic Approaches in Forensic Analysis

The first step for forensic analysis is to collect forensic data. In case multiple different data sources exist, the examiner collects data of different formats generated by applications or operating systems. Even though several tools support forensic analysis, the heterogeneous formats of collected data used for investigation can hold some challenges for further analysis.
In this section, we focus on existing approaches to integrate Semantic Web technologies in the process of forensic analysis and assist the investigator in analyzing digital evidence.

One solution we found is an ontology-based approach for a forensic analysis of data collected from mobile devices presented by Wolf [2013] and Alzaabi et al. [2013]. The overall goal of both works is to handle large amounts of data gathered from mobile devices of different formats more easily by using an ontology and using the structured presentation for forensic analysis. Wolf [2013] examines the forensic of various structures of mobile devices. Therefore, the author introduces an ontology for forensic analysis. He aims to automatically draw conclusions by the correlation of results. Alzaabi et al. [2013] develop a layered ontology-based framework, which builds a new forensic analysis tool for content retrieved from Android smartphones. Inference engines and classification mechanisms use that digital evidence process in order to identify new implicit information. The framework consists of five main layers:

1. **Evidence Space**: This layer holds potential evidence objects, such as files, videos, and images.

2. **File Wrappers**: The Wrapper is a simple program, which extracts descriptive information from files in the evidence space, such as the MIME type.

3. **File Description and RDF/OWL databases**: The file wrapper uses the file description database. The next layer uses the RDF/OWL database for concepts and relationships.

4. **Concept and Relationship Extraction**: This layer extracts concepts from the previous file description database. For example, data obtained from a mobile device would be a contact that belongs to the 'Contact' class, an image that belongs to the 'Media' class, and a Word document file that belongs to the 'Document' class. In addition, the layer maintains the relationships between concepts.

5. **Domain and Application Ontologies**: These two layers collaboratively form an ontological model for a particular environment. This model consists of concepts (or classes) and the relationships among them. For instance, in a smartphone environment, the framework considers message, person, email, and an event as individual domain ontologies.

Dosis et al. [2013] also uses ontologies for representing and integrating digital evidence. The goal is to provide partially or eventually fully automated analysis of the large volumes of digital data by parsing evidence into their semantic representation automatically. One example the paper describes is the forensic analysis of storage media, such as hard disks, USB sticks, and SD cards.
The automation consists of the following steps:

1. Collecting of relevant data and transforming it into their semantic representation and generating semantic assertions.

2. An OWL reasoning engine infers conclusions on semantic data based on created assertions.

3. An investigator, who uses SPARQL, formulates and executes queries against the integrated set of data.

As an example, the solution should analyze a forensic disk image. The first step asserts an image file and declares it as a member of the class 'ImageFile'. In the second step, the OWL reasoning engine infers that it is also a member of its superclass 'MultimediaFile'. The last step performs further queries concerning the image file.

Cuzzocrea and Pirrò [2016] also presents a framework that aims to integrate the advantages of *Semantic Web* technologies. The goal is to create a knowledge base that an analyst can consult to gain insights from previous cases. Thereby, the authors aim to lift digital investigations to the level of knowledge-driven digital investigations. By coupling foundational ontologies with domain-specific ontologies, the framework achieves abstractions in order to create a modular and knowledge-driven approach to digital forensics.
Furthermore, the knowledge base should complement digital forensic tools. The growth

of cyber-crimes and the increased usage of information and communication technologies trigger that approach. The framework consists of four layers:

1. **Knowledge**: This layer collects foundational ontologies that model general concepts.
2. **Integration**: The use of the Resource Description Framework (RDF) integrates the collected data.
3. **Reasoning**: This layer accesses the previously integrated data.
4. **Querying**: In this layer, the user can query integrated and reasoned data.

For example, attack ontologies integrate firewall logs and the output of collected traffic from Wireshark into semantic data. SPARQL can then query the data which the framework represented semantically. The queries aims to identify malicious communication or suspicious events logged by the firewall.

4.5 Semantic Representation of Log Data

In the following section, we present types of log data which were represented semantically by ontologies. We also describe the purpose of data integration using semantic ontologies.

Wolf [2013] and Alzaabi et al. [2013] both describe an approach to represent logs from mobile devices for analysis purposes. We described details of their solutions in Section 4.4. Their main goal is to overcome heterogeneous log formats for any forensic analysis.

There are several approaches for defining ontologies for other logged data such as logs of network traffic, application logs, and logs of user activities. Several works, presented in this section use these computer logs for security-related forensic analysis. The work of de Souza Nascimento et al. [2011] discusses an approach of using Semantic Web and an ontology to analyze security logs with the goal to identify possible security issues. The defined ontology should improve the search for patterns and evidence of certain attacks. The paper uses logs generated from *Web Application Firewall* for their ontology, which are logs generated by the program *ModSecurity*[8].

Nimbalkar et al. [2016] describes another approach which classifies log data semantically. The author discusses a framework, which automatically parses log data in various formats and generates a semantic description of their schema and content in RDF. Using regular expressions, the parser separates the log entries into columns and rows for a dictionary-based classification. In addition to Nimbalkar et al. [2016], whose work describes parsing any sort of data of any format, Holliday et al. [2017] focuses on using RDF for defining an ontology describing a log format for grid environments in order to overcome different formats of logged data. Furthermore, Clark et al. [2004] deals with the effective scalability

[8] https://modsecurity.org, accessed: 2019-06-04

of the mass of data that accumulates during the investigation of event information. This data should be made manageable by semantically strong representational models and automated methods of correlating such event data.

To summarize, the main goal of described solutions is to present log data semantically in order to gather a structured representation of the data. Therefore, heterogeneous log formats should be overcome. Furthermore, Semantic Web technologies should make an enormous amount of unstructured event log data scalable.

CHAPTER 5

Semantic Models for Log Data Representation

In this chapter, we define the used data models to semantically represent file-system activities. Furthermore, we define additional semantic models to represent *File Access Events* and background knowledge. Moreover, we explain the relation between file-system log entries and file access events and give insights on how we transform log entries into access events. Lastly, we present our concept of how events are related to each other in order to reconstruct a file life-cycle.

5.1 Concept Architecture

We aim to explore a new approach via a near real-time forensic analysis. Therefore, we need technologies, techniques, and external tools that enable us to achieve the following conceptual points. The concept aims to fulfill our requirements which we defined in Section 2.1:

1. **Read log data in near real-time:** Log data is read via a near real-time system in order to avoid handling huge amounts of collected data. This requires a service on the analyzed hosts that provides live file system logs.

2. **Overcome log formats of different file systems:** We aim to jointly analyze different log formats. This requires a unified semantic vocabulary for log data from different file systems.

3. **Analyze log data:** As soon as required log data has been collected, we perform the analysis. This requires an automatic parsing and filtering of relevant log data.

4. **Trace file activities:** For the tracking of file activities, we will only focus on file system log data of macOS[1]. We aim to trace all operations of a file. Furthermore, all operations of a single file are related to each other in order to achieve a file's life-cycle. This requires an algorithm that relates subsequent events to trace past activities of the same file. In addition, we require a user interface that enables us to display the files life-cycle as a graph.

In summary, in order to meet the defined requirements, we need an architecture that connects: a logging service, a tool that parses and filters log data, a tool for transforming log data into semantic data, and a service that saves the data, retraces past file activities, and reconstructs a files life-cycle.

5.2 Specification of Semantic Models

Klas and Schrefl [1995] describe *Semantic Models* as the following:

> *Semantic Models* are a tool to capture the meaning of data by integrating relational concepts with abstraction concepts. This approach aims to provide high-level modeling primitives. The use of these high-level models helps to facilitate the representation of real-world situations.

As we introduced in Section 1.3, our goal is to semantically represent file system log data in order to analyze the history of logged file activities. Furthermore, an analysis of the collected data should be provided in regards to suspicious activities that might result in data exfiltration. We specified the following sequence of actions which we need to achieve in order to fulfill our goals:

1. **Transform and Correlate**: In this step, we transform file-system log entries into the associated high-level file access event, e.g. a move operation of a file into a different location is transformed into the access event *Moved*.

2. **Correlate Events**: We need to correlate identified file access events chronologically in order to reconstruct the life-cycle of a file.

3. **Identify User**: We have to resolve the user who performed recorded file activities in order to know the responsible real-life person behind any suspicious activities.

4. **Identify Channel**: In order to detect suspicious patterns, we have to identify the target channel of file operations. For example, in case the target location of a move operation is not an internal channel, we suppose that a user moved the file outside the excepted borders and exfiltrated data.

[1] macOS High Sierra version 10.13.6

Consequently, we require a semantic model that allows us to represent the content of file system log data and assist in the analysis process. Furthermore, the system has to link background information with recorded log data in order to reason about who performed logged activities and to detect suspicious actions such as copying a file to an exfiltration channel.
We need several ontologies in order to create the required vocabulary which enables us to create an RDF representation of file system logs, file access event, and needed background knowledge. Therefore, we define the following semantic models:

1. **File System Event Model**: This model represents raw file-system event data received from the operating system. We describe more details about the event data in Section 3.3. Section 5.3 explains the model itself.

2. **File Access Event Model**: This model illustrates the high-level event performed to a file, which can be a copy operation or a rename of a file. Section 5.4 describes the model.

3. **User Account Model**: This model represents a user whose actions are monitored by examined file-system logs. The user account helps to reason about the real-life person who performed file actions. The model is part of the defined background knowledge. Section 5.5 explains the model.

4. **Exfiltration Channel Model**: This model comprises information on communication channels that a user can use to exfiltrate data. Exfiltration channels are part of our background knowledge as well. Section 5.5 contains more details about the model.

We describe each model in more detail in the following subsections and clarify relations between those models.

5.3 File System Events Data Model

As we described in Section 5.2, the first semantic model represents file-system events. We collect those events in log data produced by auditing services of the underlying operating system. Therefore, we require an RDF representation of file events including information about the file accessed, the system call performed and who performed it at what time.

We described existing approaches to semantically represented log data in Section 4.5. Existing solutions mainly focus on various security logs, firewall logs, and logs produced by mobile applications. However, we were not able to find any ontology for representing file-system log data. Consequently, we create a new semantic ontology. Therefore, we extend a more general class *LogEntry* introduced by Ekelhart et al. [2018] that represents a single generic log entry, as well as the *Host* and *Logtype* classes, shown in Figure 5.1.

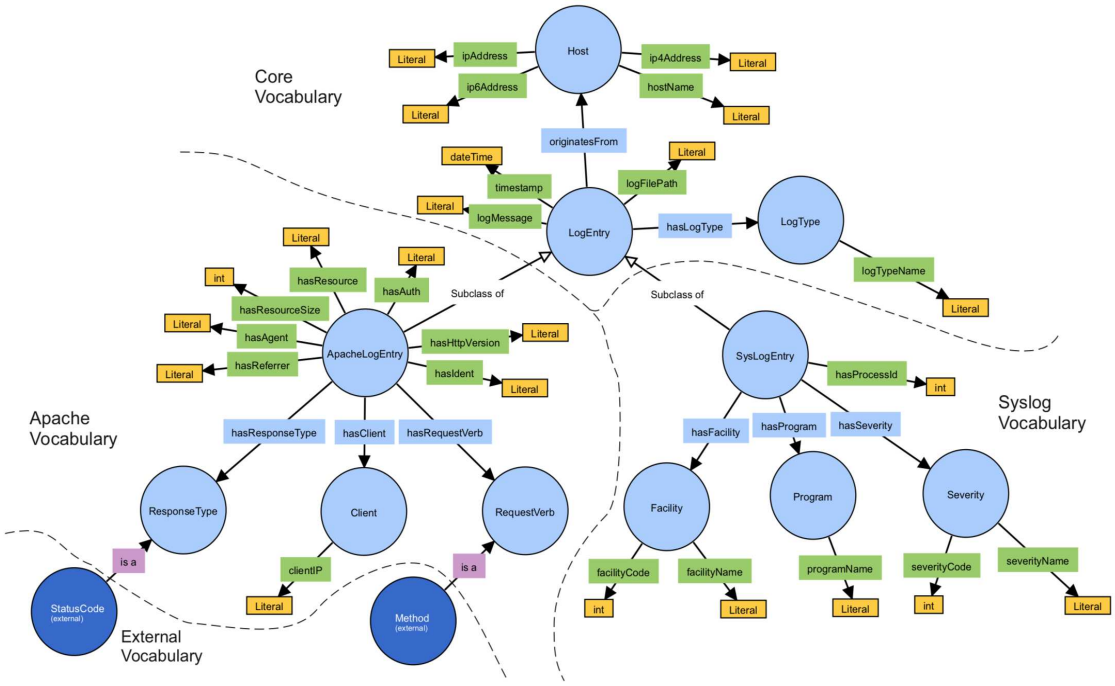


Figure 5.1: Log vocabulary presented by Ekelhart et al. [2018]

Windows and Unix file-system logs contain slight differences. In Unix, the property *accessCall* in auditd file system logs contains the performed system call. However, Windows Event Viewer logs presents this information by the properties *eventID* and *accessInfo*. We identified the distinction by comparing log data produced by an auditing service from OpenBSM[2] for file operations on macOS and Windows logs from the internal Event Viewer. Kurniawan et al. [2019b] also describes the semantic representation of Windows Log Events.

Due to this differentiation, we extend the class *FileSystemLogEntry* further to subclasses *UnixFileSystemLogEntry* for Unix specific file system logs and *WinFileSystemLogEntry* for Windows specific log data. This thesis focuses on macOS[3] file system log data, therefore we use the type *UnixFileSystemLogEntry* for any further explanations and analysis.

Figure 5.2 shows the vocabulary which presents the subclass *FileSystemLogEntry* of super-class *LogEntry*. A *FileSystemLogEntry* always relates to a *File*, containing the pathname of the accessed file. Furthermore, the log entry has a *User* who performs file activities. The field *timestamp* tells when the file operation happened. Lastly, the field *accessCall* contains the performed system call.

[2]http://www.trustedbsd.org/openbsm.html, accessed: 24-03-2020
[3]macOS High Sierra version 10.13.6

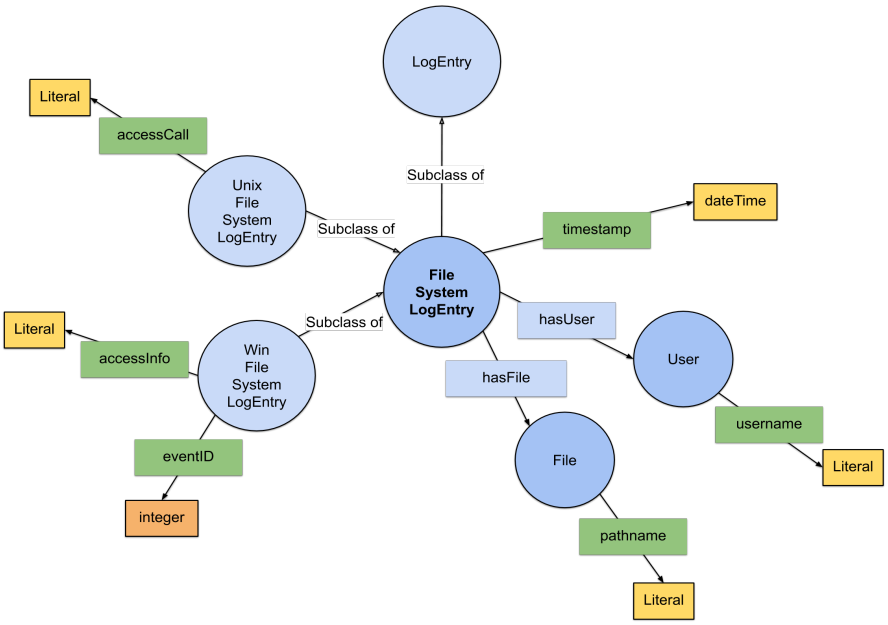


Figure 5.2: Vocabulary for File System Logs

5.4 File Access Events Data Model

As we described in Section 5.2, file access events are the high-level representation of corresponding file-system logs. The events represent the actual access activity of a file. Figure 5.3 displays the used ontology. A *FileAccessEvent* always relates to a source *File* and a target *File*. The source file represents the original file, whereas the target file represents the resulting file after a file activity. For example, in case of a *rename* file operation the target file contains information on the new filename. The class *User* gives information about the domain and username of the involved user account. In addition, the source *Host* and target *Host* displays information of the client which performs the file activity. The class *Action* represents the type of the file access event performs. Figure 5.4 displays the ontology of a *FileAccessType*. We distinguish six different types of events. Table 5.1 presents the instances of these types. A *FileAccessType* consists of a label and a comment, providing a more detailed description of the actual event.

As we can see in the table we do not distinguish between the types *Create* and *Modified*. The reason for this is that the corresponding system calls of both access events intersect with each other. In addition, we model a *delete* file operation as access type *MoveToRecycleBin*, which represents a *move* file activity into the directory */.Trash/*.

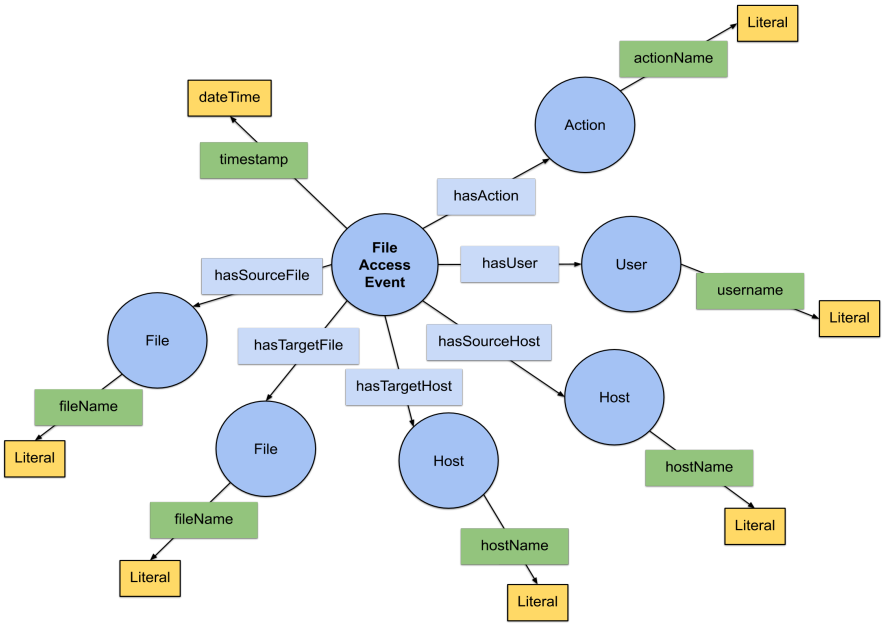


Figure 5.3: Vocabulary for File Access Events

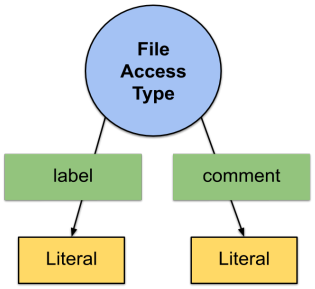


Figure 5.4: Vocabulary for File Access Type

Label	Comment
Created	Created a new file
Created/Modified	Created a new file or modified an existing file
Created/Copied	Created a new file or copy a file from the same or different locations
Renamed	Renamed a file
Moved	Moved a file from one location to another
MovedToRecycleBin	Moved a file to Recycle Bin / Trash

Table 5.1: Instances of File Access Type

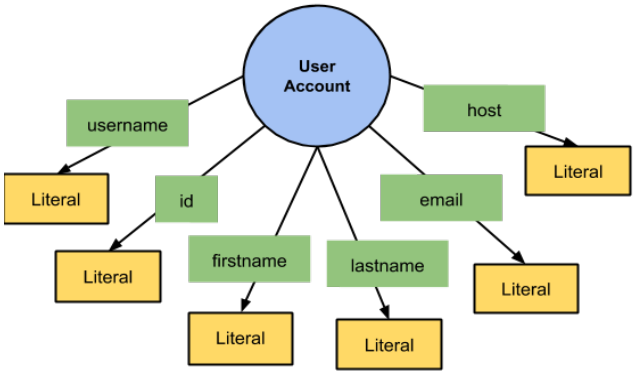


Figure 5.5: Vocabulary for User Accounts

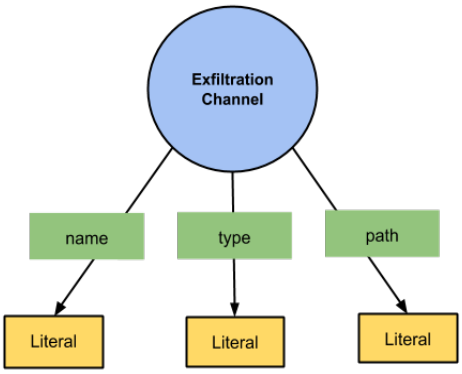


Figure 5.6: Vocabulary for Exfiltration Channels

5.5 Background Knowledge Data Model

The background knowledge consists of information about the user account associated with an event and the channel of the file activity.

Figure 5.5 presents fields of the class *UserAccount*, which are *username*, *id*, *firstname*, *lastname*, *email*, and *host*. The model is helpful in case several monitored user accounts are connected to the same person. This makes it possible to query for all operations by a specific person, irrespective of the specific account used. Figure 5.6 displays the vocabulary of an *ExfiltrationChannel*. The ontology gives meaning to extracted file paths. Thereby, the implemented service can distinguish *copy* or *move* operation performed to an external channel. It also recognizes if users move or copy files internally to another directory within its acceptable borders.
The property *type* defines if the specific channel is *internal* or *external*. An internal channel is any path that is considered to be within the companie's boundaries. The service considers other paths, which are not internal channels, as external channels.

5.6 Relations between Semantic Models

A *FileAccessEvent* is the high-level representation of one or multiple instances of the class *FileSystemLogEntry*. Therefore, we require a transformation process that converts a specific pattern of log entries into a file access event.

In case a *FileAccessEvent* consists of only one *FileSystemLogEntry*, we can directly map file, host, and timestamp information from the log data to the related event. However, in case of a copy operation, a *FileAccessEvent* consists of multiple log entries. In this case, we require information on the original file from one log entry, which will become the source file of the access event. In addition, we require data about the new file from another log entry, which will become the target file of the access event. We describe the exact implementation of this process in Chapter 6.

5.7 File Life-Cycle Reconstruction

In this section, we present our approach to reconstruct the history of file activities. *File Access Events* are, after their construction, single events that are independent of each other. In order to relate subsequent events we introduce a process that constructs those relationships. Thereby, we focus on linking successive events, and aim to retrace file transformations into new files, e.g. in case a user copies a file a new file is created. After a transformation, a new file life-cycle starts. We require to link the history of the original file and the history after the file transformed.

We aim to achieve the reconstruction via a *SPARQL Construct* query. Figure 5.7 illustrates our concept of the construct query. As shown in the figure, the query consists of three optional patterns, containing four events. The patterns are optional on their own, due to the fact that a life-cycle does not have to contains all patterns. Each pattern illustrates a case which we require to consider. Figure 5.7 contains the overview of each case. Figure 5.8 shows examples of those patterns and events.

The cases are the following:

1. **Link successive equal events:** In this case, the service has to link two successive events with the same source and target pathnames. The first section of Figure 5.8 illustrated this case. Both events represent a *Created_Modified* which happens after each other.

2. **Transformations:** In case of a file transformation a new pathname will be created, e.g. on *Moved*, *Renamed*, *Created_Copied* and *MovedToRecycleBin* events. Thereby, the service has to consider the different source and target pathnames. The second section of Figure 5.8 shows two types of transformations. In the left example an *Created_Modified* event is followed by an *Moved* event. The move operation transforms the target pathname. The right example shows two transformation events in sequence. In both examples, the target pathname of the first event is

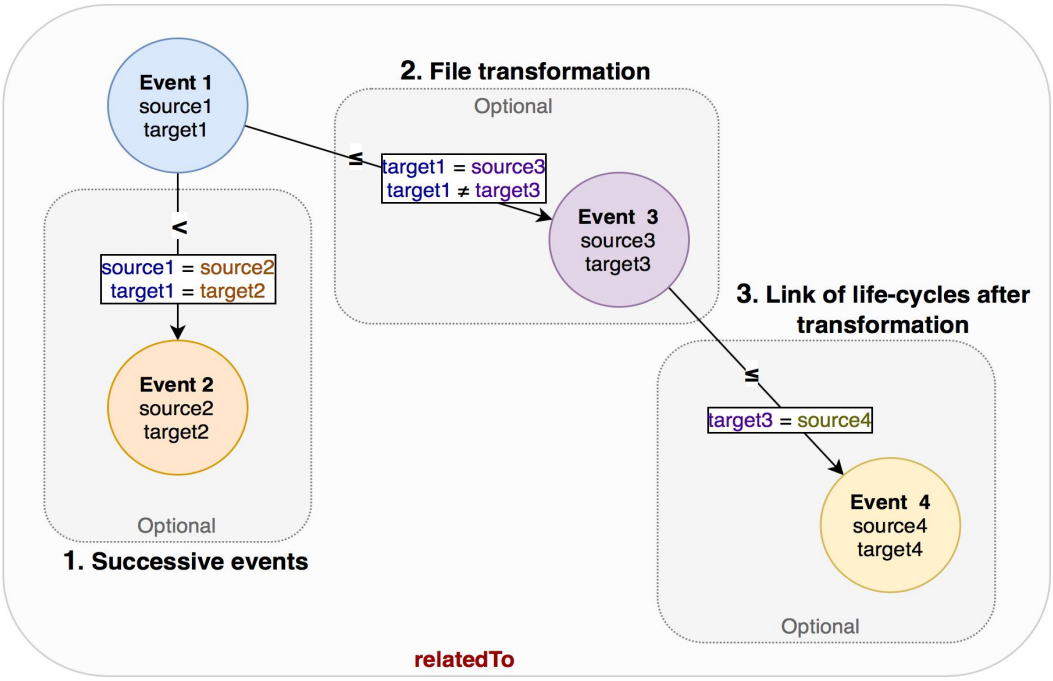


Figure 5.7: Conceptual draft to construct *relatedTo* property between events

equal to the source pathname of the second event. However, the target pathnames of both events differ.

3. **Link life-cycles after a transformation:** After a transformation a new file life-cycle starts. However, the previous life-cycle is also related to the new history, containing activities of the transformed file. Therefore, we have to filter for a pattern containing a fourth event which represents the first event of a new file life-cycle. The third section in Figure 5.8 shows the linkage of two life-cycles. The previous cycle terminates by an *Moved* event. The newly created life-cycle starts with event *Created*.

By performing the presented *SPARQL Construct* querying over all saved semantic data we link all events. The query adds the property *relatedTo* between each event. This enables us to query for relations of a certain pathname, which will then deliver all events which belong to the files life-cycle. We describe further details about the implementation in Chapter 6.

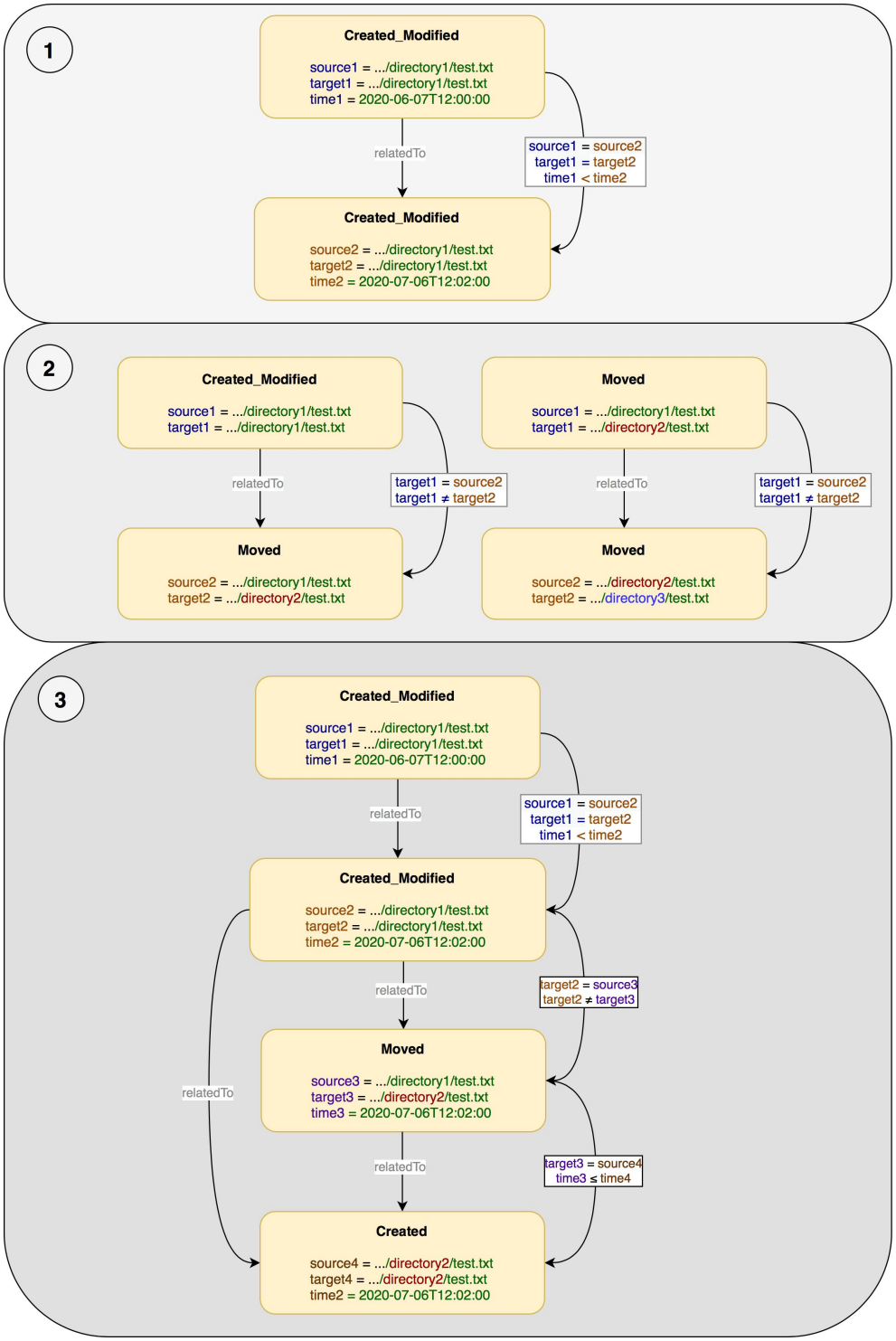


Figure 5.8: Example patterns considered by the construct of the *relatedTo* property between events

CHAPTER 6

Implementation

In the previous chapter, we defined our architectural concept, the semantic model, and the required vocabulary. In this chapter, we discuss details about our prototype to implement the introduced approach. This includes an overview of the overall architecture, details about involved components, performed configurations, and preconditions.
In addition, we describe used external tools and their adaptions to our needs, and communication between those systems. We further discuss why we chose specific technologies alongside their advantages and disadvantages.

6.1 Architecture

The prototype implementation in this work is a software solution that enables a near-real-time analysis of collected semantically represented data. The result of the automated semantic analysis is a file life-cycle presented as a history graph. We illustrate the architecture of the software solution in Figure 6.1.

During the development, we focused only on macOS file-system logs. Thereby, any external tools selected required to support macOS. However, we considered providing a solution that can easily be adapted to other Unix-based operating systems.

The prototype system includes a server component that receives file-system log data, transforms received data into its semantic representation according to our defined ontology, and relates access events. The architecture shows, from left to right, the process from a log record, from its creation until analysis and storage. The process consists of the following main steps:

1. **Filter File-System Event**
   The first component is the client which produces log data of file-system events. The

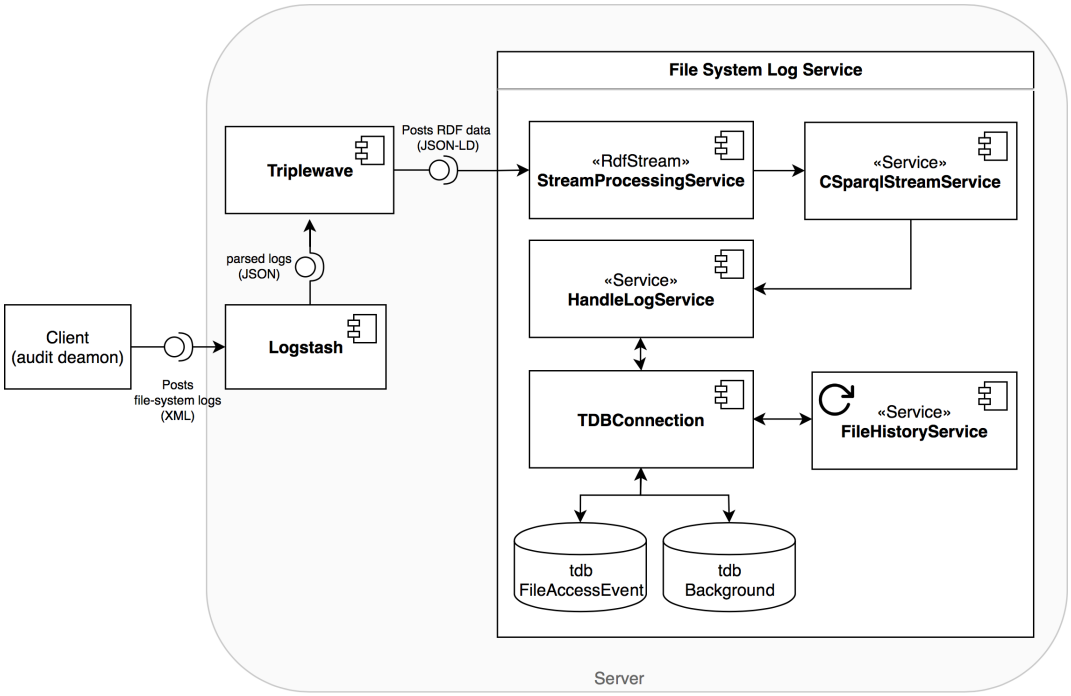


Figure 6.1: Architecture Diagram of prototype

client runs a Bash script which uses the auditing service of *OpenBSM*[1] for collecting logs. We present more details about the extraction of log data in Section 6.4. The script sends the log data via an HTTP Post request as XML records to *Logstash*. We describe *Logstash* in Section 6.5.1. This tool parses the raw log data and filters out irrelevant information. The component discards unneeded log records and filters only for records which the later analysis requires. Thereby, the amount of log data is scaled and easier manageable during the semantic analysis process. Logstash forwards data in JSON format via a *WebSocket* to a *TripleWave* instance, which we introduce in Section 6.5.2.

2. **Transform into Semantic Data**
   We lift filtered log entries into RDF data by applying the defined *FileSystemLogEntry* ontology. An instance of *TripleWave* performs the transformation automatically. We present the semantic model in Section 5.3. After the tool represents the collected log data semantically, *TripleWave* forwards the RDF data as a stream via a *WebSocket* to the implemented service *"File System Log Service"*.

3. **File Access Event Processing**
   In this step, *C-SPARQL* constructs *File Access Events* from an received RDF log data stream. The provided service handles the incoming data stream of semantic

[1] http://www.trustedbsd.org/openbsm.html

data by the components *StreamProcessingService* and *CSparqlStreamService*. The service *StreamProcessingService* extends an *RdfStream* and receives incoming JSON-LD data from the *WebSocket*. The service transforms the data into triples and inserts them into an instance of an RDF stream. The service *CSparqlStreamService* holds both the RDF stream and initializes an instance of a *C-SPARQL* engine. Furthermore, it handles multiple instances of *C-SPARQL* result proxies. We require a separate result proxy for each file access event type. Each proxy references a SPARQL query that the engine executes over the RDF stream. Each query searches for a different pattern of file-system events and constructs the associated *File Access Event*. Section 6.6 explains the construction in more detail. In case a query, registered in a proxy, produces an output within the current window, the engine forwards the result to service *HandleLogService*, which handles all found C-SPARQL results. The service takes over the triples of the *File Access Event* and hands them over to the component *TDBConnection*, which handles connections to triple stores. We use *Apache Jena's TDB* component to persist data. Section 6.2 presents the component.

4. **File History Reconstruction**
   Lastly, the server component relates the saved *File Access Events*. Moreover, the system processes related events to a file life-cycle graph. Section 6.7 describes the process.

Based on the principles of design science as our research methodology, we required several iterations of the design cycle in order to finalize the used technology stack and implemented algorithms. We explained the applied methodology in Chapter 2. Due to our requirements concerning the compatibility of the selected tools with each other and by the support of the underlying operating system we required reiterations of the design cycle. We needed to test the service for extracting log data by its outcome and compatibility in order to meet requirements for the later software system. Thereby, the service required to deliver certain information on performed file activities, which the component needed to reconstruct a file live-cycle. Furthermore, we evaluated several options available, which we describe in Section 6.4. Moreover, we developed multiple versions of the service *"File System Log Service"* until requirements were met. This includes adaptations on *C-SPARQL* construct queries, in order to filter for access events. We describe the rationale behind defined construct queries as well as discovered difficulties in Section 6.6. Furthermore, we altered the build process of a file live-cycle repeatedly. Section 6.7 describes the final implementation details about the reconstruction.

In the following section we explain the technology stack of our software solution as well as the communication between implemented components and their relation between each other.

6.1.1 Implementation Details

The technology stack of the implemented prototype consists of various external tools coupled with a software developed in this thesis.

To build a Semantic Web application, we used the open source Java framework *Apache Jena*[2] and included its standard and core libraries. *Jena* provides a programmatic environment for *RDF*, *RDFS* and *OWL*, *SPARQL* and includes a rule-based inference engine. Furthermore, we included the components *ARQ* and *TDB*. *ARQ* provides a *SPARQL 1.1* query engine, and *TDB* offers libraries in order to create a native triple store. We explain further details about the triple store and the used *Apache Jena* component in Section 6.2 *Apache Jena TDB Component*.

Moreover, in order to process a continuous flow of RDF data, we incorporated an implementation of a *Continuous SPARQL* engine called *C-SPARQL*[3]. Therefore, we integrated dependencies for core components of the *C-SPARQL* engine and added the component *rsp-services-api*[4], which is a Java API to access a RSP service implementation in combination with the *C-SPARQL* engine.

Furthermore, we used the framework *Spring Boot*[5] and *Apache Maven*[6] in order to manage the application configuration and the build process.

The source code of the implemented prototype and other implementations can be found on a public GitHub repository [Fröschl, 2020].

6.2 Apache Jena TDB Component

In this section, we are going to explain the component used for handling the RDF storage and the processor used for performing SPARQL queries on this storage.

The component *TDBConnection* provides access to triple stores of semantic data. It uses version *TDB1* of the *Apache Jena* component *TDB*[7]. The library provides an API for creating RDF models and datasets. We create datasets by calling the static method *createDataset("<path to directory>")* of class *TDBFactory* and provide the path to the directory where we want to create the TDB for indexes and node tables. This method call creates a *TDB-backed dataset* for which we can access its model[8].

The *TDB* component offers the possibility to work with transactions that enabled us to perform concurrent read and write operations in the same store. Moreover, it follows

[2] `https://jena.apache.org`, accessed: 2019-05-06
[3] `http://streamreasoning.github.io/TripleWave/docs.html`, accessed: 2019-05-16
[4] `https://github.com/streamreasoning/rsp-services-api#rsp-services-api`, accessed: 2019-05-06
[5] `https://spring.io/projects/spring-boot`, accessed: 2019-05-06
[6] `https://maven.apache.org`, accessed: 2019-05-06
[7] `https://jena.apache.org/documentation/tdb/index.html`, accessed: 2019-05-06
[8] `https://jena.apache.org/documentation/tdb/java_api.html`, accessed: 2019-05-06

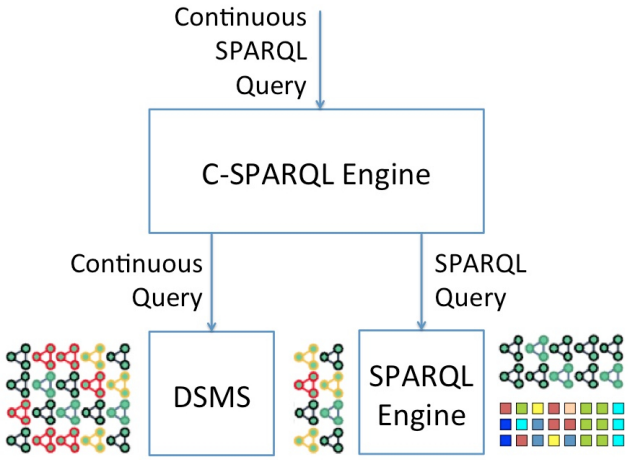


Figure 6.2: C-SPARQL engine architecture

a *Multiple Reader* or *Single Writer* (MRSW) policy. This allows us to save new data within its triple store and query for semantic data at the same time.

Furthermore, in order to query for semantic data using the query language *SPARQL* we integrate the query engine *ARQ*, which is a *Apache Jena* SPARQL processor[9].

We found that the open-source framework *Apache Jena* is an easy to use tool to handle semantic data programmatically. Moreover, the framework is widely used, and therefore a variety of examples on development forums and articles are available. Furthermore, the website of *Apache Jena* also provides clear and structured documentation that simplifies our implementation efforts.

6.3 C-SPARQL as Complex Event Processing Language

C-SPARQL allows us to introduce *Semantic Complex Event Processing* (SCEP) into our prototype. We explain the component *C-SPARQL* which we use in order to query over a continuous stream of incoming RDF data received from *TripleWave*. The display the data flow in the architecture in Figure 6.1.

Generally, a *C-SPARQL* engine consists of two sub-components , as illustrated in Figure 6.2. The component *DSMS* is responsible for executing continuous queries over RDF data streams. It produces temporal RDF snapshots, which will be the input for the *SPARQL Engine*, which runs a standard SPARQL query against it. The binaries of the C-SPARQL Engine use *Esper* and *Apache Jena-ARQ*[10].

[9] `https://jena.apache.org/documentation/query/index.html`, accessed: 2019-05-06

[10] `https://www.w3.org/community/rsp/wiki/RDF_Stream_Processors_Implementation`, accessed: 2019-05-06

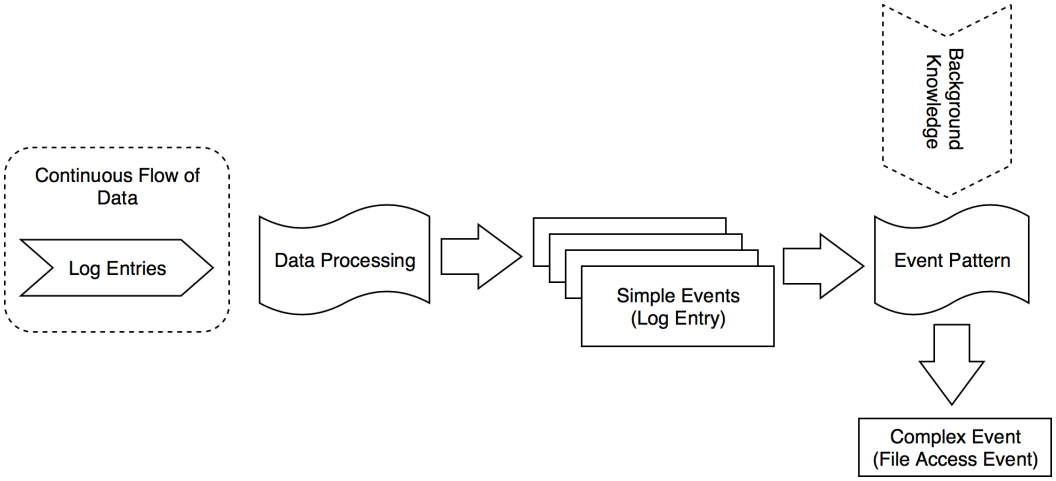


Figure 6.3: Complex Event Processing: flow from *Log Entry* to *Complex Event*

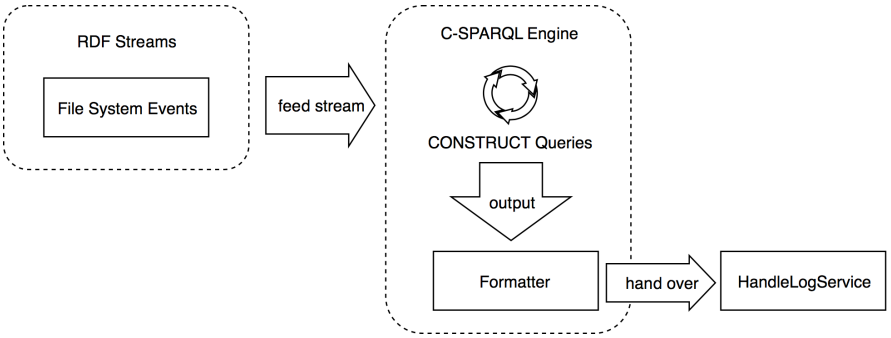


Figure 6.4: Process of C-SPARQL engine to handle RDF streams

Schaaf et al. describe *Complex Event Processing* as tool to break an event down into its constituent events. The authors describe an event as complex if it is to be analyzed in a more detailed granularity.
In our case, a *File Access Event* defines a complex event. Multiple *Log Entries* compose a single *File Access Event*, combined by additional background knowledge. Figure 6.3 shows the flow from *Log Entries* until we receive a *Complex Event*. The flow processes a continuous flow of data, containing *Log Entries* and transforms each entry into a *Simple Event*. In the next step, we filter for patterns within these events. Found patterns result in *Complex Events* which we combine with additional *Background Knowledge*.

In the prototype implementation we use *C-SPARQL* construct queries in order to detect *File Access Events*. When querying over the continuous flow of data we consider time windows. We define an individual *C-SPARQL* construct query for each access type. These queries include filters concerning access calls and other parameters contained in log entries. Figure 6.4 shows the process executed by our implementation. Incoming RDF streams are fed to the C-SPARQL engine, which executes registered queries over

incoming data. Service *CSparqlStreamService* holds an RDF stream for incoming file system log data.
Furthermore, it holds an instance of a *CsparqlEngine*, which registers queries for each access type. For each query we defined a *Formatter* that catches results of the cosponsoring query and forwards the result to the *HandleLogService*. Section 6.6 provides further details regarding the registered queries.

We chose *C-SPARQL* because of its ability to define windows over an RDF stream and its support to register multiple queries over a single RDF stream. In addition, we were able to use all features provided by SPARQL, since *C-SPARQL* is an extension of the query language SPARQL [Barbieri et al., 2010b]. Also, the number of implementation examples and discussions available was in favor of using *C-SPARQL*.

6.4 File System Event Extraction

In this section, we focus on concepts concerning the extraction of file-system log data. We examined several approaches to collect log data. Since we focus only on macOS file-system logs, the following section contains alternatives to view file activities on macOS.

The operating system macOS offers several options to audit file events. Tools such as the internal software *Instruments* or the command *fs_usage* present similar outputs, containing data about the used access call, program, timestamp, and file pathname. However, knowledge about the related user is missing in the produced log output of both tools.

Fs_usage [11] is a terminal command and represents a popular troubleshooting tool that filters file operations based on the mode provided when running the command. The use of tag *-f* defines the mode. For example, when filtering for only network-related file events, we can use mode *network*. Mode *pathname* outputs only events related to the given pathname.

Instruments [12] is an integration of *Xcode*[13] and offers powerful analysis tools with a graphical user interface. The software offers a variety of different types of metrics that support the gathering of information about file system reads, writes, and other operations. The software is built upon the dynamic tracing framework *DTrace* [Gregg and Mauro, 2011].

DTrace [14] was created by *Sun Microsystems* and intended to troubleshoot kernel and application problems. It offers powerful options to trace file activities. In order to interact

[11] `https://ss64.com/osx/fs_usage.html`, accessed: 2019-05-06
[12] `https://developer.apple.com/library/archive/documentation/Performance/Conceptual/PerformanceOverview/PerformanceTools/PerformanceTools.html`, accessed: 2019-05-06
[13] `https://developer.apple.com/xcode/`, accessed: 2020-01-16
[14] http://dtrace.org/blogs/about/, accessed: 2019-04-30

with *DTrace*, we can create scripts using the D programming language, similar to *C*, and *awk*. However, it supports also a direct usage on the command line. According to Gregg and Mauro [2011], *DTrace* is used to observe exactly how the file system responds to applications, how effective file system tuning is, and the internal operations of file system components.
Even though DTrace offers a powerful tool that met our requirements, we decided to use the auditing service by *OpenBSM* to produce our log data. Reasons for this decision were primarily the easy use of *OpenBSM*, compared to the rather complex *DTrace* syntax. Also, the produced outputs contained all information needed.

OpenBSM [15] is an open source implementation of *Sun's Basic Security Module* (BSM) security audit API and file format. It can be used to maintain system audit streams[16]. The implementation enabled us to perform live auditing on macOS in order to detect file system events such as opening or editing of a file. The *audit daemon* (*auditd*) offers a wide arrange of system log data. Therefore, we have to configure the output in order to filter only for records concerning file-system operations. The configuration file, located in path */etc/security/audit_control*, provides information on what *auditd* is currently auditing. The file contains information on the configured audit classes and for which the operating system produces user log data. Provided audit classes are found in a file located at */etc/security/audit_class*. We configured file related audit classes concerning read, write, access, modify, create, and delete operations. Appendix A in Section 8.3 shows our configuration files for setting up the *audit daemon*.

The tool saves logged content in the directory */var/audit*. The files are binary, therefore we use additional tools to read its content. The tool *auditreduce* provides filters for records and tool *praudit* handles conversions of their content in a readable format, such as XML by using the *-x* flag.

The macOS kernel reports system events in real time. A process with sufficient privilege can access the resulting records by reading the named pipe /dev/auditpipe [Gehani and Tariq, 2012]. By combining provided tools, we achieve to extract real-time file system log data by the command in Listing 6.1.

```
1 $ sudo auditreduce -o file='<directories to audit>' /dev/auditpipe |
    praudit -xn
```

Listing 6.1: Command to output real-time log data in XML format

In summary, the tool *auditreduce* provides real-time log data and offers the possibility to filter for only events occurring within given paths defined by the *-o* flag and parameter *files*.The tool *praudit* can presents its output in XML format by using an *-x* flag.

[15] `http://www.trustedbsd.org/openbsm.html`, accessed: 2019-04-30
[16] `https://github.com/openbsm/openbsm`, accessed: 2019-04-26

6.5 External Tools

In the upcoming sections, we introduce external tools used within our prototype system. These tools include *Logstash* and *TripleWave* to filter, parse and transform logged data into RDF. *Logstash* receives and parses the extracted log data from client machines and forwards the filtered data to *TripleWave*. The tool transforms received data into its semantic representation. The system architecture contains both components in Figure 6.1.

6.5.1 Logstash

The tool *Logstash* is a free and open-source component. It was originally developed by Jordan Sissel and now maintained by the *Elastic* team. On the *Elastic*'s website[17] *Logstash* is described as an open-source, server-side data processing pipeline that ingests data from a multitude of sources simultaneously, transforms it, and then sends the data to your favorite *stash*. It provides a framework for log collection, centralization, parsing, storage, and search [Turnbull, 2016]. Furthermore, we can use the filter component in order to manipulate and filter data. In addition, *Logstash* can then transmit data over different output plug-ins to a subsequent system, which further handle parsed log data.

The configuration file *auditpipe.conf* of our file-system log pipeline consists of tree sections [Turnbull, 2016]. Listing 6.2 illustrates those sections.

```
1 input{}
  filter{}
3 output{}
```

Listing 6.2: Tree sections of a Logstash pipeline

The *input* section describes how the pipeline receives file system audit records. The *filter* portion contains any transformation steps and the *output* portion contains the configuration of how the tool transmits the data. Our implementation of file *auditpipe.conf* can be found in our GitHub repository [Fröschl, 2020]. However, the following segments give a short overview of the operations performed on each log entry.

In the *input* portion of our *auditpipe.conf* we described how the pipeline receives data for parsing and further transformation activities.
Logstash provides an input plug-in called *Auditbeat*[18] which delivers all user activities and processes them without the need of using *auditd*. However, this plug-in is only available for *Linux* and is not compatible to macOs's version of *auditd* provided by *OpenBSM*. According to an open feature request[19] on the *Elastic* Github repository, the support for Mac auditing is planned but currently not implemented. Therefore, we chose to use *OpenBSM* instead in order to get log data directly from the auditing service.

[17] `www.elastic.co`, accessed: 25-01-2020
[18] `https://www.elastic.co/de/beats/auditbeat`, accessed: 2019-05-17
[19] `https://github.com/elastic/beats/issues/6061`, accessed: 2019-05-17

We received log data from clients via HTTP. Therefore, we use the input plug-in *http* for incoming data. The *filter* portion contains any data manipulations and transformations of the received data. The auditing service of OpenBSM outputs more data than we actually need to handle. Therefore, we filter log events for specific *accessCalls*, which are related to file operations we want to find. By excluding log entries of specific *accessCalls* we also aim to make the amount of logged events more manageable and we reduce the amount of log data that a subsequent system needs to process. Moreover, we extract information about the timestamp, file path, filename, hostname, IP address, and the username from log entries. Furthermore, we associate each log entry with a UUID. The plug-in *date* helps us to convert the timestamp into *UTC*. The plug-in *xml* extracts information contained in XML-tags and the plug-in *mutate* helps us to rename fields or remove unused data.

The *output* portion defines any plug-ins for sending transformed data to a particular destination. We are using the plug-in *websocket* which runs a *WebSocket* server on default port *3232* and publishes any messages in JSON format to all connected *WebSocket* clients.

6.5.2 TripleWave

TripleWave[20] is an open-source framework for creating RDF streams and publishing them over the Web. The framework is able to process data streams into RDF and publishes it again as an RDF stream, which then the RDF stream processing engine can consume.

TripleWave was motivated due to the lack of standards concerning protocols and mechanisms for RDF stream exchange, which limits the adoption and spread of RSP technologies on the Web [Mauri et al., 2016]. Communication supported by *TripleWave* is a pull-based consumption of stream data and push communication through *WebSockets*. Thereby the tool can pull any data from *Logstash* and push it again via a *WebSocket* for our prototype to consume. *TripleWave* transforms data from *Logstash* into its semantic representation according to our defined semantic models introduced in Section 5.3.

In order to illustrate how we involved *TripleWave*, we added a simplified version of its architecture in Figure 6.5, which is based on the overall architecture presented by Mauri et al. [2016]. As we can see in the figure, *TripleWave* receives JSON data from *Logstash* and converts the data into an RDF stream. Any data received is non-RDF and *TripleWave* transforms it to its RDF representation. We define the semantic model by an *R2RML*[21] mapping file.

Mauri et al. [2016] describes that the tool uses a generic transformation process. A *R2RML* mappings specifies this process. Although these mappings were originally conceived for relational database inputs, *TripleWave* uses light extensions that support other formats such as CSV or JSON. After the conversion, *TripleWave* pushes data via a *WebSocket* as JSON-LD[22] and our client prototype consumes the data. We downloaded

[20] `http://streamreasoning.github.io/TripleWave/docs.html`, accessed: 2019-05-16
[21] `https://www.w3.org/TR/r2rml/`, accessed: 2019-05-15
[22] `https://json-ld.org`, accessed: 2019-05-05

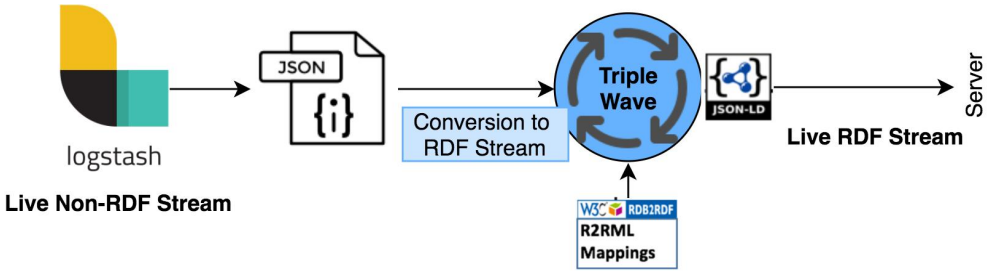


Figure 6.5: The architecture of TripleWave

the *TripleWave* project from Github[23] in December of 2018, therefore any of the following issues described are concerning the implemented version at this time. We faced an issue during the transformation of data to its RDF representation. When using the original transformation process the first subject is set as the subject for all following objects. Therefore the resulting RDF data is not accurate in case we have to transform a more complex semantic model. In order to solve the transformation we changed the implementation of function *transform* within two files which were *r2rml-js/r2rml.js* and *stream/enricher.js*. Appendix B in Section 8.3 shows our implementation of both files.

After we added the required changes to the transformation implementation, we had to create two additional components:

1. a web stream connector (shown in Listing 6.3) and
2. a *R2RML* mapping file containing the semantic model (shown in Listing 6.4).

For their implementation we followed the documentation on the *TripleWave* homepage[24].

In our connector we implement a *nodejs Transform Stream*, which reads incoming data of the *Logstash WebSocket*. The presented stream produces a stream of all file event log data.

```
1  var stream = require('stream');
   var util = require('util');
3  var WebSocket = require('ws')

5  var Transform = stream.Transform ||
   require('readable-stream').Transform;
7
   function LogstashStream(options) {
9      if (!(this instanceof LogstashStream)) {
           return new LogstashStream(options);
11     }
```

[23] https://github.com/streamreasoning/TripleWave, accessed: 2019-05-15
[24] http://streamreasoning.github.io/TripleWave/docs.html, accessed: 2019-05-16

```
    this.socket = new WebSocket('ws://0.0.0.0:3232/');
    var _this = this;

    this.socket.on("message", function incoming(data) {
        console.log(JSON.parse(data));
        if (!_this.close) {
            _this.push(JSON.parse(data));
        } else {
            _this.push(null);
        }
    });
    // init Transform
    Transform.call(this, options);
}

util.inherits(LogstashStream, Transform);
LogstashStream.prototype._read = function(enc, cb) {};
LogstashStream.prototype.closeStream = function() {
    this.close = true;
};
exports = module.exports = LogstashStream;
```

Listing 6.3: Nodejs Transform Stream

In order to transform the produced stream into its RDF representation, we needed the *R2RML* mapping which we display in Listing 6.4.

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix sr: <http://purl.oclc.org/rsp/srml#>.
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix time: <http://www.w3.org/2006/time#>.
@prefix schema: <https://schema.org/>.
@prefix fileSystem: <http://w3id.org/sepses/vocab/fileSystemLog#>.
@prefix : <http://epfl.ch/mapping/>.


:LogEntryMap a rr:TriplesMap;
rml:logicalSource [
rml:root true;
rml:source "jdbc coso"; ];
rr:subjectMap [ rr:template "http://w3id.org/sepses/vocab
                        /fileSystemLog#LogEntry-{id}"];
rr:predicateObjectMap [ rr:predicate fileSystem:accessCall;
rr:objectMap [ rr:template "{accessCall}" ]];
rr:predicateObjectMap [ rr:predicate fileSystem:timestamp;
rr:objectMap [ rr:template "{timestampLog}" ]];
```

```
rr:predicateObjectMap [ rr:predicate fileSystem:logMessage;
rr:objectMap [ rr:template "{logMessage}" ]];
...
rr:predicateObjectMap [ rr:predicate fileSystem:hasFile;
rr:objectMap
[ rr:template "http://sepses.ifs.tuwien.ac.at/
    vocab/fileSystemLog#File-{id}";
rr:parentTriplesMap :FileMap ]];
...

:FileMap a rr:TriplesMap;
rml:logicalSource [
rml:root true;
rml:source "jdbc coso"; ];
rr:subjectMap [ rr:template "http://sepses.ifs.tuwien.ac.at/
    vocab/fileSystemLog#File-{id}"];
rr:predicateObjectMap [ rr:predicate fileSystem:fileType;
rr:objectMap [ rr:constant "_:n3" ]];
rr:predicateObjectMap [ rr:predicate fileSystem:pathname;
rr:objectMap [ rr:template "{pathname}" ]].

..
```

Listing 6.4: R2RML mapping of file events

The *R2RML* file for audit records contains only an abstract from the entire configuration file in order to illustrate how our project defines triples.
The first *TripleMap* (line 12 - 31) defines the class of a log entry with its properties. At line 27 we define the *LogEntryMap* as parent map of a subsequent *FileMap*, that contains file properties. The entire file can be found in our GitHub repository [Fröschl, 2020].

6.6 Event Detection and Semantic Data Analysis

In the following section we describe our conceptual steps to detect and create *File Access Events*. Therefore, we need to clarify which file operations we want to detect and which system calls the file system carries out for each event. For this purpose, we collected and analyzed real file event log data to identify the underlying access calls for each event, i.e.:

(i) **Move**, (ii) **Copy**, (iii) **Create**, (iv) **Modify**, (v) **Rename** and (vi) **Delete**.

(i) **Move** represents a move of a file into a different directory. The terminal command `mv` will trigger the access call `rename(2)`. Thereby, the operation changes the pathname of the file. However, in the case of a move operation, only the directory differs from the original pathname. The name of the file doesn't change.

(ii) **Copy** is an operation that creates an identical file based on a copied file. When a user copies a file, we discovered a sequence of two access calls. The first event is

the access of the original file and the second event represents the creation of the new file. However, the performed access calls differ between a copy operation into the original folder or into a different directory which we need to consider when trying to detect those patterns. In addition, produced access calls can also differ by the sequence of access calls executed, in case a user copies a file manually in the *Finder* or by a Bash command `cp`. The *Finder* triggers the access call `open(2) - read`, which opens the file to copy, followed by the call `setattrlist()`, which sets the attributes of the copied file. The terminal command `cp` triggers the access call `fstatat(2)`, followed by the call `open(2) - write,creat,trunc`.

(iii) **Create** operations can trigger a collection of different access calls. The program determines which call the operation triggers in order to access a resource. Therefore, the access call depend on the particular program or process. Therefore, the same *File Access Event* can result in multiple different access calls by the operating system and for full coverage, we would have to consider all possible system calls by any program a user has access to. This is based on the fact that system calls are provided by the operating system via an application programming interface. Programs access those interfaces in order to request needed resources from the kernel [Silberschatz et al., 2018].

(iv) **Modify** is similar to a *Create* operation. Triggered access calls can be a collection of system calls depending on the used program. The access calls produced by *Modify* and *Create* also overlap. Therefore these activities are hard to differentiate and the service combines them into one event called *Created_Modified*.

(v) **Rename** is an operation that changes the file name. This activity triggers the same access call as *Move*. However, the directory remains the same, and only the file name changes. Therefore, we need to consider the path of the source and target file in order to distinguish between *File Access Events*.

(vi) **Delete** represents a *Move* operation in the directory */.Trash/* within macOS. Therefore, in order to detect this activity, we need to find a log entry which shows a *Move* operation into a specific directory. We do not consider a final deletion from */.Trash/* in this thesis. However, this pattern would trigger another access call which the *C-SPARQL* engine needs to consider.

We aim to detect the pattern of each file operation which consists of a single log entry or a sequence of two log entries. Furthermore, each pattern results in the creation of a new *File Access Event*. We construct the event by the attributes contained in the detected log entry or entries.
Table 6.1 summarizes which file activities we monitor, which access calls the file activities trigger and we needs to consider, and lastly, which *File Access Event* should be constructed.

The column *File System Operation* represents defined file system operations we aim to identify. The next column shows any *access calls* contained in logged audit records.

File System Operation	**macOS Access Call**	**Access Event Type**
Move	rename(2)	Moved
Copy (into different directory)	*access of file to copy:* fstatat(2) *create new file:* open(2) - write,creat,trunc	Created/Copied
Copy (in same directory)	*access of file to copy:* open(2) - read *create new file:* setattrlist()	Created/Copied
Create	open(2) - read,creat, open(2) - write,creat	Created
Modify	openat(2) - read, open(2) - read,write, open(2) - write,creat,trunc, open(2) - read,write,creat,trunc, open(2) - write,creat, open(2) - read,write, open(2) - write	Created/Modified
Rename	rename(2)	Renamed
Delete	rename(2) –> move to directory */.Trash/*	MovedToRecycleBin

Table 6.1: Mapping of macOS access calls to FileAccessEvent actions

Finally, the column *Access Event Type* displays the associated high level *File Access Event*.

In the prototype system, we are using the complex event processing engine *C-SPARQL*. The engine registers a *SPARQL* construct query for each pattern. Each query filters for attributes which are for example the access call file pathname, directory, and the timestamp. In the case of a *copy* operation, the query correlates subsequent log entries according to the contained timestamp.

Figure 6.6 shows an overview of the filters used for constructing *File Access Events* such as: (i) **Created**, (ii) **Created_Modified**, (iii) **Renamed**, (iv) **Moved** and (v) **MovedToRecycleBin**. .

(i) We detect **Created** by a combination of two access calls. The operating system creates those access calls solely when the user generates a new file.

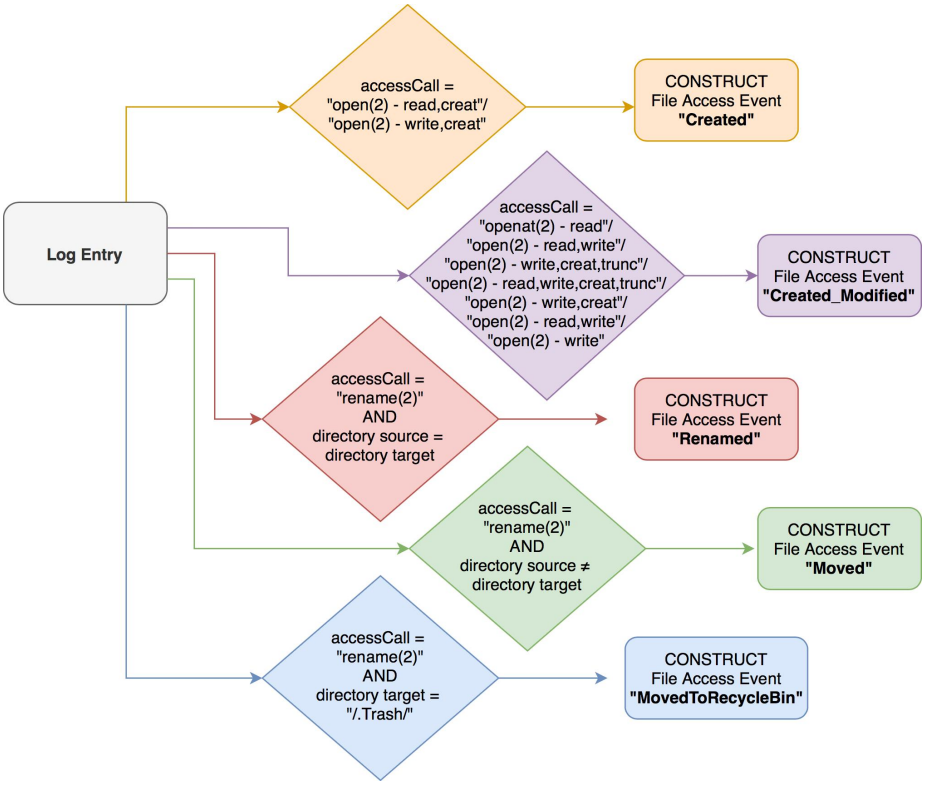


Figure 6.6: File Access Types constructed from a single log entry

(ii) The system detects **Created_Modified** by a collection of different access calls. The access call performed depends on the program used for modifications. Filtered system calls mostly contain information on operations such as *write*, *read*, *creat*, and *trunc*. Therefore we are not able to distinguish between a *Create* or *Modify* operation in most cases.

(iii) **Renamed** triggers access call *rename(2)*. Moreover, we require to control the directory of the source and target pathname, which have to be equal.

(iv) **Moved** also performs access call *rename(2)*. It differentiates by *Renamed* by the filter for the source and directory pathnames. The paths have to be different on a move operation.

(v) **MovedToRecycleBin** is a special case of a *Moved* event. Thereby, the target path has to be the directory */.Trash/*.

In addition to the previously explained *File Access Events* we define two separate *SPARQL* construct queries for the event **Created_Copied** displayed in Figure 6.7 and Figure 6.8. The first query identifies copy activities within the same directory and the second query searches for copy operations into another directory. The defined queries only find copy

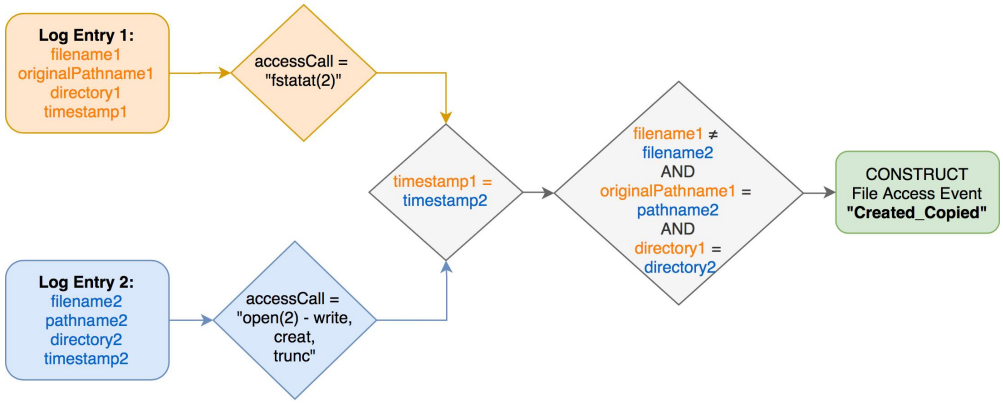


Figure 6.7: Construct of File Access Event *Created_Copied* in the same directory

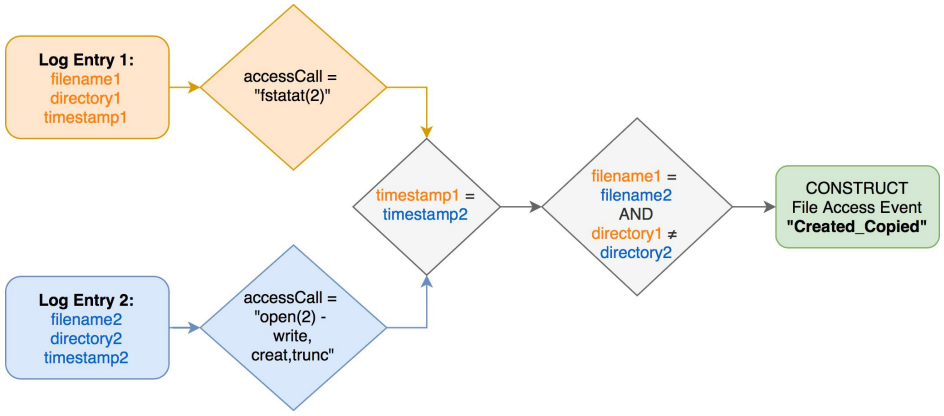


Figure 6.8: Construct of File Access Event *Created_Copied* to a different directory

operations produced by the Bash command `cp`. In case a user performs copy operations manually in the *Finder*, or by any other process, we have to define additional queries since the pattern differs from the pattern the Bash command `cp` triggers.

In Figure 6.7 the first log entry (*Log Entry1*) represents the access of the file, which the user aims to copy. The second log entry (*Log Entry2*) shows the actual *write* or *create* operation of the new file. Both events have to occur at exactly the same time. On macOS, a copied file always will contain the original file name with an additional string *"copy"* at the end of the name. Therefore, the new pathname is equal to the original pathname of *Log Entry1*. However, the *filename* of the new file changes. Moreover, we also check if the directory is the same in both events.

Figure 6.8 shows the pattern of log entries triggered for copy actions into a different directory. This pattern differs from the previous sequence by the filename and the directory. The filename of the new file stays the same as the original filename. However, the directory differs between both events.

In summary, in order to transform audit log records to file access events, we have to detect, interpret, and correlate the underlying system calls. In addition, the service has to consider different log entry patterns based on the used process or program. In case different patterns result in the same *File Access Event*, we have to define multiple queries, one for each pattern, since the pattern of access calls depends on the program used. Thereby, we are able to construct a *copy* event by different patterns, i.e. in case a user performs a copy via a terminal command or manually in the *Finder*.

6.7 File History Graph

One of our main goals is to reconstruct a file life-cycle in order to retrace past operations. We explained in Section 5.7 how we aim to link related events. In summary, we aim to link *File Access Events*, which belong to the same history by the property *relatedTo*. Therefore, we perform a *SPARQL* contruct query which filters for the sequence of four events (i.e. Event1, Event2, Event3 and Event4), displayed in Figure 5.7. We explained the concept behind the query and our rationale in Section 5.7. In case such a pattern is found the query constructs the triples *Event1 relatedTo Event2*, *Event1 relatedTo Event3*, *Event1 relatedTo Event4*, etc. and vice versa. Figure 5.8 illustrates an example containing all four events, which we describe in Section 5.7.

Furthermore, we require to perform the construct query for every occurred pathname. Consequently, we need to call the query recursively. Before each iteration, the service has to adapt the initial pathname of *Event 1* to the next occurred pathname. We solved the recursive call of the construct query programmatically. Therefore, we implemented the service *FileHistoryService*, which is also included in the architecture of our system in Figure 6.1.

The service *FileHistoryService* collects all occurred pathnames and executes the construct query for each. Since we are aiming to implement a near real-time auditing system we perform the construction of property *relatedTo* by a scheduler that continuously performs our queries. Figure 6.9 shows a flowchart of the tasks performed by the implemented scheduler, since we experienced fast results on this interval. We choose a period of 60 seconds for the scheduler. We experimented with a period of 60 seconds for the scheduled task. However, we would suggest making the period configurable in order to be able to adapt the time for life-cycle reconstructions. The scheduled task constructs the property *relatedTo* for all past *File Access Events*.

In order to query for a file life-cycle, we implemented a single-page Web App providing a search box for file pathnames. We built the front-end via Vanilla JS[25] in order to create a Single Page Application. The front-end communicates with the back-end server via AJAX. The service offers an HTTP endpoint to receive any POST requests containing the pathname for which the system needs to build the history graph.

[25] `https://www.sitepoint.com/single-page-app-without-framework/`, accessed: 2019-06-03

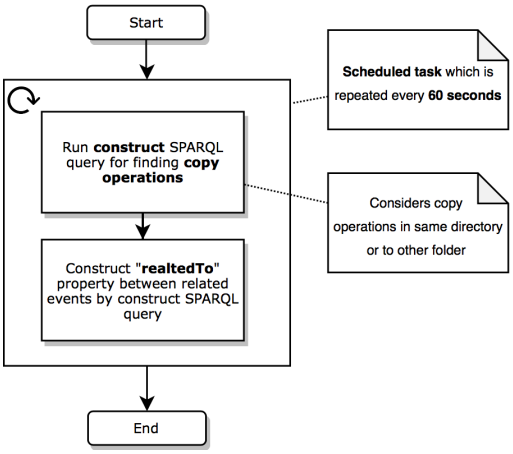


Figure 6.9: Flow of scheduler task in service *FileHistoryService*.

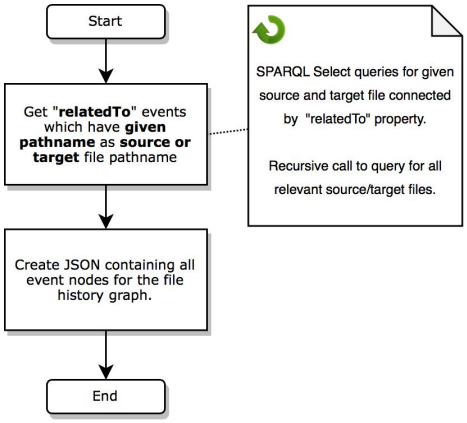


Figure 6.10: Process of creating JSON Nodes of a file-history by service *FileHistoryService*.

We use the JS library *vis.js*[26] in order to visualize a graph. Therefore, the server has to transform the history data as JSON, which an endpoint then hands over to *vis.js*. The JS creates all nodes and edges contained in the JSON. Nodes represent pathnames and edges show performed actions. The service *FileHistoryService* constructs the JSON. Figure 6.10 shows the process of creating the JSON files containing the history graph.

To provide more information on the operation, we added a click listener to each edge. By clicking on any edge of the graph, more information on the selected action is shown. The shown details include the timestamp when the user performed the action, the source, and target pathname as well as the involved user. An example of the resulting graph is shown in Figure 6.12.

[26] `https://visjs.org`, accessed: 2019-06-03

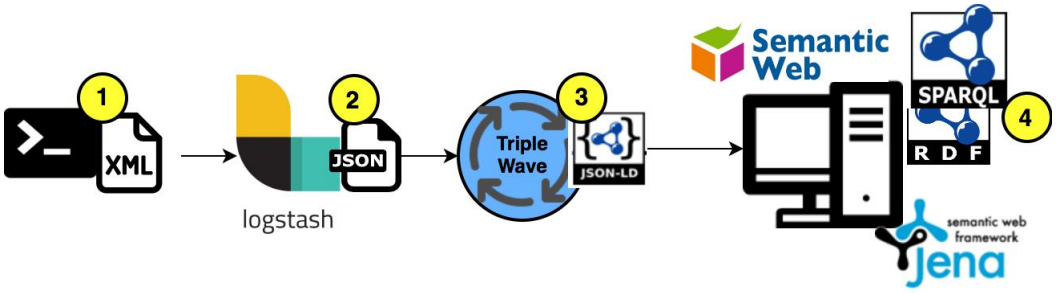


Figure 6.11: Event flow overview of all components

6.8 Event Flow of User Interaction

In this section, we aim to provide an overview of the messages passing through the components. We created a single *move* event resulting in an access event *Moved*. We displayed each message created in the solution in Figure 6.1. The presented example only shows a single log entry which results in one complex event.

1. An event flow starts with a user interaction. The user accessed a file and moved it to another directory. Listing 6.5 displays the resulting audit record XML. To make the messages easier to present, we only presented one log entry.

```
<record version="11" event="rename(2)" modifier="0"
   time="Sat Feb  8 14:09:53 2020" msec=" + 602 msec" >
   <path>/Users/Agnes/Desktop/test/testfile.txt</path>
   <path>/Users/Agnes/Desktop/test/testfile.txt</path>
   <attribute mode="100644" uid="501" gid="20" fsid="16777220"
   nodeid="55968680" device="0" />
   <path>/Users/Agnes/Desktop/sample/testfile.txt</path>
   <path>/Users/Agnes/Desktop/sample/testfile.txt</path>
   <subject audit-uid="501" uid="501" gid="20" ruid="501"
   rgid="20" pid="370" sid="100008" tid="50331650 0.0.0.0" />
   <return errval="success" retval="0" />
</record>
```

Listing 6.5: Step 1: Audit record of move operation

2. *Logstash* parses the created XML record. The tool performs defined filters in the pipeline configuration file and outputs a JSON shown in Listing 6.6.

```
{ pid: '370',
  logMessage:
   '<record version="11" event="rename(2)"...> ... </record>', ...
  logTypeName: 'UnixAuditdFile',
  host: '128.130.233.117',
  pathnameTarget: '/Users/Agnes/Desktop/sample/testfile.txt',
  hostName: 'e233-117.eduroam.tuwien.ac.at',
  ip: '128.130.233.117',
  dirSource: '/Users/Agnes/Desktop/test/',
  dirTarget: '/Users/Agnes/Desktop/sample/',
  accessCall: 'rename(2)',
  timestampLog: '2020-02-08T14:09:53.000Z',
  id: 'b6c0fb05-b510-4fad-bc79-37a6555835a2',
  username: '501',
  pathnameSource: '/Users/Agnes/Desktop/test/testfile.txt'
}
```

Listing 6.6: Step 2: JSON output from Logstash

3. In this step, *TripleWave* transformed the received data into its RDF representation. Listing 6.7 presents the resulting JSON-LD.

```
{ "@graph": [
    { "@graph": [
        {
        "@id": "http://w3id.org/sepses/vocab/fileSystemLog#
          File-b6c0fb05-b510-4fad-bc79-37a6555835a2",
          "http://w3id.org/sepses/vocab/fileSystemLog#dirSource":
          "/Users/Agnes/Desktop/test/",
          "http://w3id.org/sepses/vocab/fileSystemLog#dirTarget":
          "/Users/Agnes/Desktop/sample/",
          ...
          "http://w3id.org/sepses/vocab/fileSystemLog#pathnameSource":
          "/Users/Agnes/Desktop/test/testfile.txt",
          "http://w3id.org/sepses/vocab/fileSystemLog#pathnameTarget":
          "/Users/Agnes/Desktop/sample/testfile.txt"
          },
        {
        "@id": "http://w3id.org/sepses/vocab/fileSystemLog#
          Host-b6c0fb05-b510-4fad-bc79-37a6555835a2",
          "http://w3id.org/sepses/vocab/fileSystemLog#hostName":
          "e233-117.eduroam.tuwien.ac.at",
          ...
          }
       ...
```

Listing 6.7: Step 3: Excerpt of JSON-LD output from *TripleWave* of move event

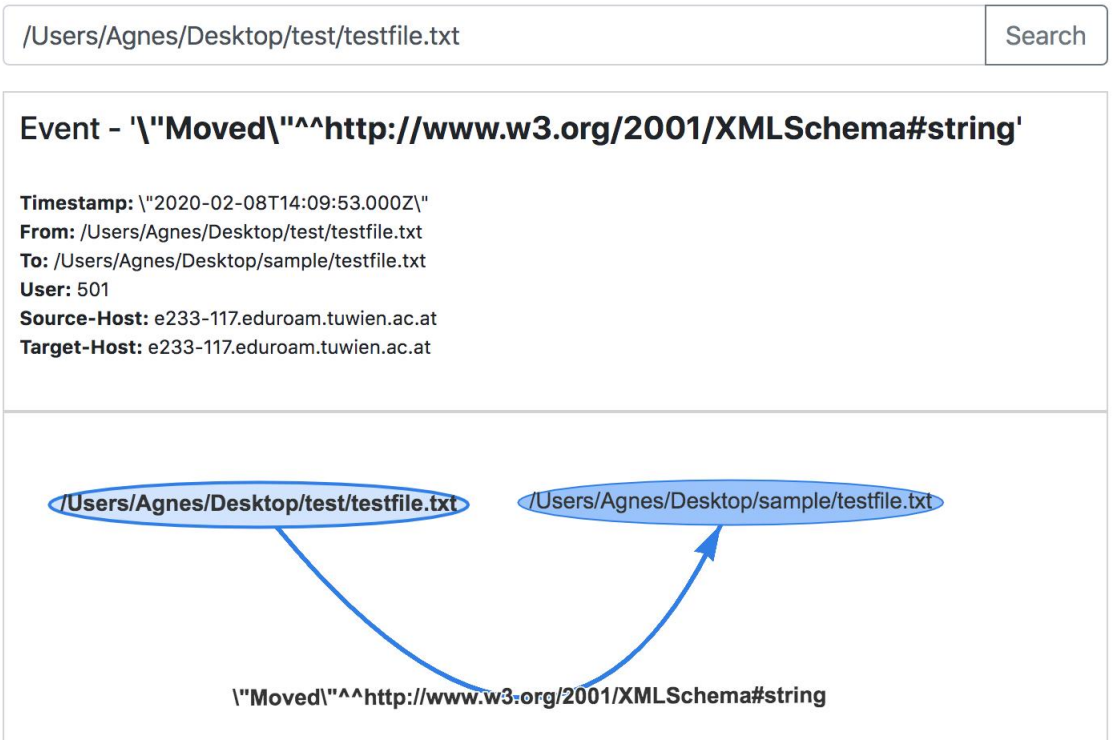


Figure 6.12: Step 4: Graph presentation of pathname in Web UI

4. In this step, our prototype analysed the consumed RDF data and built a history graph of all occurred filepaths. The user can then view the created graph by a Web UI shown in Figure 6.12.

5. In the last step, we link to defined *Background Knowledge* which enables us to reason about the communication channel and the user who performed the audited file activity. Figure 6.13 and Figure 6.14 show an instance of our *Background Knowledge* regarding a *User Account* and an *Exfiltration Channel* with the retrieved information from the log data.

The displayed example contains only one access event. When the user selects an edge of the graph, the UI displays information on the event.

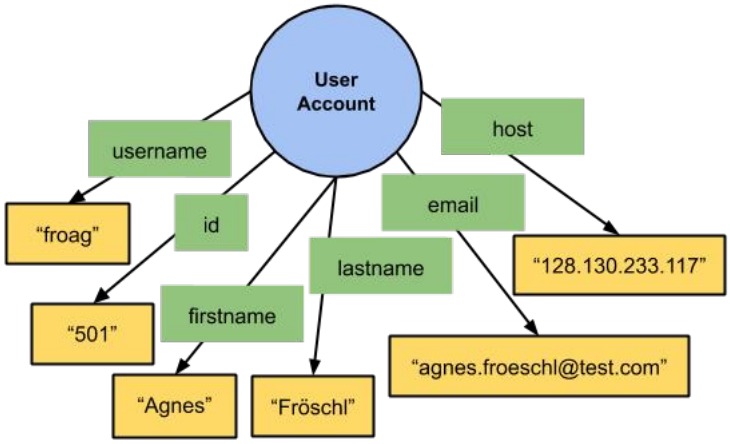


Figure 6.13: Step 5: Instance to Background Knowledge *User Account*

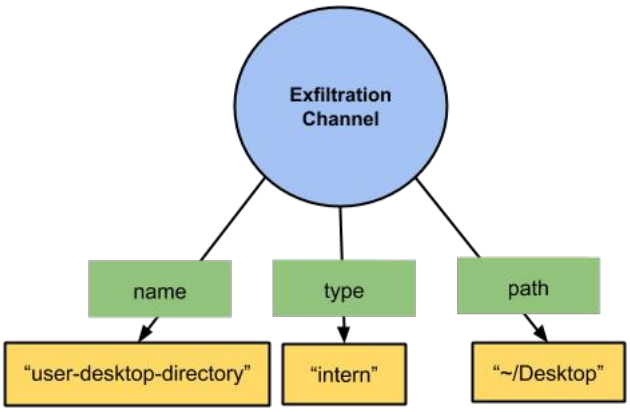


Figure 6.14: Step 5: Instance to Background Knowledge *Exfiltration Channel*

CHAPTER 7

Evaluation

In this chapter, we present several scenarios that aim to evaluate our implementation. First, we describe automated scenarios to verify the detection capabilities on single and mixed file operations. Then we present the performance results and discuss them. Furthermore, we present a data exfiltration scenario with manual user interaction and measure the performance in this realistic setup. Lastly, we conceptually compare our approaches to existing forensic analysis tools, which we defined in Chapter 4, and discuss alternative approaches to *C-SPARQL*.

7.1 Automated Scenarios

The following sub-sections define the conducted evaluation of automated scenarios. This includes a definition of how we measure test results, as well as details about the test setup and how the evaluation simulates file operations.

Automated test runs aim to evaluate our first research question described in Section 1.3. This includes the question under which circumstances we can represent log data semantically by a near-real-time system. Therefore, we aim to evaluate the performance of our prototype system.

7.1.1 Measurement Factors

In our evaluation, we aim to measure the performance and throughput regarding detected events of the prototype system. We use the practices of information retrieval in order to calculate the performance of detected events [Goutte and Gaussier, 2005]. The indicators are *precision* and *recall* which we calculate for each test run.

- **Precision:** This is the fraction of the relevant detected events among all detected events.

		Predicted class	
		detected	not detected
Actual class	detected	True Positive (TP)	False Negative (FN)
	not detected	False Positive (FP)	True Negative (TN)

Table 7.1: Confusion Matrix

Metric	**Formula**
Precision	$\frac{TP}{(TP+FP)}$
Recall	$\frac{TP}{(TP+FN)}$

Table 7.2: Metrics with Formula

- **Recall:** This is the fraction of the total amount of produced events that we detected.

Table 7.2 displays the formula of both calculations. Furthermore, our classification model, which the formulas use, is shown in the confusion matrix in Table 7.1. The **Predicted class** describes all file events that our solution detected. The **Actual class**, on the other hand, represents the actual events that happened on the computer.

The following list contains all cases of our classification model in Table 7.1:

- **True Positive (TP):** A TP includes detected events which actually happened on the computer.
- **False Negative (FN):** A FN includes not detected events that actually happened on the computer.
- **False Positive (FP):** A FP includes detected events that did not happen on the computer. This includes additional events that do not fit in predicted event patterns.
- **True Negative (TN):** A TN would be an event which we did not detect and that did not happen on the computer. This metric does not make sense in our scenario, and therefore we will not include it in our evaluation. Thereby, we are also not able to calculate the *accuracy*.

In summary, in order to calculate the performance of detected events we classify detected *File Access Events* by our confusion matrix (Table 7.1) and calculate *precision* and *recall* using formulas (Table 7.2) for each test run. Thereby, we aim to examine scenarios by

the success rate of detected events and find limitations concerning the recognition of events. We explain the exact scenarios of our test runs in the following section.

7.1.2 Test Scenarios

In this section, we describe the scope of automatically performed test scenarios. The aim of these scenarios is to evaluate the throughput of detected events of the developed prototype system. Thereby, we also aim to identify the limitations of our system regarding event detection.

We classify test scenarios into shorter tests, which should show the overall performance regarding the detection of single event types, and longer test iterations which should show if the testing period influences results regarding the detection. Hence, we defined the following scenarios:

1. **Separate performance tests:** This scenario focuses on testing the detection of each file operation separately. We only produce events of a single file operation on each iteration. The first entry in Table 7.3 summarizes the setup. The test runs last for 5 minutes and involve a single client.

   In order to find the limits of our system, we decrease the waiting time between file activities until the system is not able to detect produced events anymore. Therefore, we start on a fixed interval of 60 seconds between events. We half the time in each iteration until we reach the limit, where we cannot detect events anymore.

2. **Mixed performance tests:** This scenario follows the same setup and focuses on the same goals as the previous scenario. However, we include a mixture of all *File Access Event* types in each test run. The second entry in Table 7.3 displays the exact setup.

3. **Load-Tests with fixed intervals:** This scenario aims to test the performance during a longer period. Hence, we are going to run the same test setup as the previous iterations including all event types. However, one iteration lasts for one hour. The third entry in Table 7.3 describes the setup. We use a fixed interval between produced events which is within the limits we detected in the previous tests. The main goal of this scenario is to identify if a longer testing period influences the performance.

4. **Load-Tests with random intervals:** In this scenario, we aim to evaluate the performance of randomly varied intervals between file operations. The test runs last for one hour and the used scripts chose the time between operations randomly. Thereby, we aim to identify if the performance changes, compared to fixed intervals between events. The fourth entry in Table 7.3 include the setup. In addition, we also aim to include the logs from up to two clients.

#	Test-Type	Sequence Events	Client	Duration	Wait times
1	Performance	Single file events	1	5 min.	60, 30, 15, 7.5, 3.75, 1.88, 1, 0.94, 0.47, etc.
2	Performance	Mixed file events	1	5 min.	60, 30, 15, 7.5, 3.75, 1.88, 1, 0.94, 0.47, etc.
3	Load testing	Mixed file events	1	1 hour	fixed time: 15 sec., 7.5 sec. and 1 sec.
4	Load testing	Mixed file events	1-2	1 hour	random times in sec.: 60, 30, 15, 7.5, 3.75, 1.88, 1

Table 7.3: Test setup

In summary, automated test scenarios aim to calculate the performance of detected events under different conditions, such as changing time interval between events, different sequences of single or mixed event types, and different durations of test runs.

7.1.3 Experimental Setup

This section describes details about the setup and the scope of automated test runs. This includes involved folders, file-types and used Bash commands in order to simulate file operations. File resources of our test runs include files of type *txt, xml, xlsx* and *docx*. Programs involved are: *TextEdit*, *Visual Studio Code*, *Microsoft Word* and *Microsoft Excel*.

In addition, we include up to 16 local folders, which contain up to 30 files with an equal number of files of each file-type. This number of available files ensures sufficient available files in case of test runs, which only contain access type *MoveToRecycleBin*.

By running a Bash script we execute the test iterations automatically. We created several scripts for running each test scenario. The scripts can be found on our GitHub [Fröschl, 2020] repository. The script picks the source and target file randomly from a list of defined directories. In order to reproduce the order of chosen pathnames and operations, we saved each iteration, in order to be able to repeat the exact same sequence of file activities. We defined a window size by the range of `[RANGE 10s STEP 3s]` in each *C-SPARQL* construct query. We discovered the best throughput of detected events on this window size during our experiments.

In Table 7.4 we list all Bash commands used for simulating each *File Access Event*. In order to simulate mouse and keyboard events, we use the command-line tool *Cliclick*[1]. This tool enables us to automatically edit a file. A modification always consists of opening the file, copying the content from the clipboard into the file, saving it and closing the file again. Table 7.4 describes the exact commands in the entry of event *Created_Modified*. When editing a file we decided to only use *txt* files and the program *TextEdit* in order to avoid problems concerning a longer start period of *Microsoft Word* or *Excel*.

[1] https://www.bluem.net/en/projects/cliclick/, accessed: 30-03-2020

File Access Event	Bash Commands
Created	touch <new-filename>
Created_Modified	open <filename>; cliclick w:1000 kd:cmd t:v ku:cmd; cliclick w:500 kd:cmd t:s ku:cmd w:1000 kd:cmd t:q
Moved	mv <original-file> <target-directory>
Renamed	mv <original-file> <new-filename>
MovedToRecycleBin	mv <original-file> <.Trash-directory>
Created_Copied	cp <original-file> <target-directory>

Table 7.4: List of used Bash commands and sequence to simulate file events.

In order to run created Bash scripts for a specific time, we installed the command-line tool *coreutils*[2] and run the scripts with the command *gtimeout*.

7.1.4 Evaluation Results

We measure the results of performed scenarios by calculating the *precision* and *recall* value for each test iteration. Table 7.3 lists the scenarios. The next two subsequent sections discuss the results and limitations of iterations lasting for five minutes and one hour.

Performance tests This section describes the results of the first two test scenarios from Table 7.3. The goal of these test runs is to measure the performance of our prototype system in a shorter period. Thereby, we aim to find the limitations regarding the detection of all *File Access Event* types by a varying throughput of events.

Figure 7.1 and Figure 7.2 show results of performed test iterations of separated *File Access Event* types. Table 7.6 includes the exact calculated numbers. As we can see in both figures the result of *precision* is always *1*. Thus, no event type caused unexpected events patterns. In order to evaluate expected event patterns, we considered any sequence of events produced by involved programs and file activities. The sequence of produced file log entries of an event type is varying, based on the used program. Section 7.1.3 describes the exact test setup and used programs.

In order to find the limit of each event type, we decreased the interval between events for each event type until the system was not able to function properly anymore and *C-SPARQL* query result become erroneous. Therefore, the amount of successfully detected events decreased significantly.

[2]http://macappstore.org/coreutils/, accessed:30-03-2020

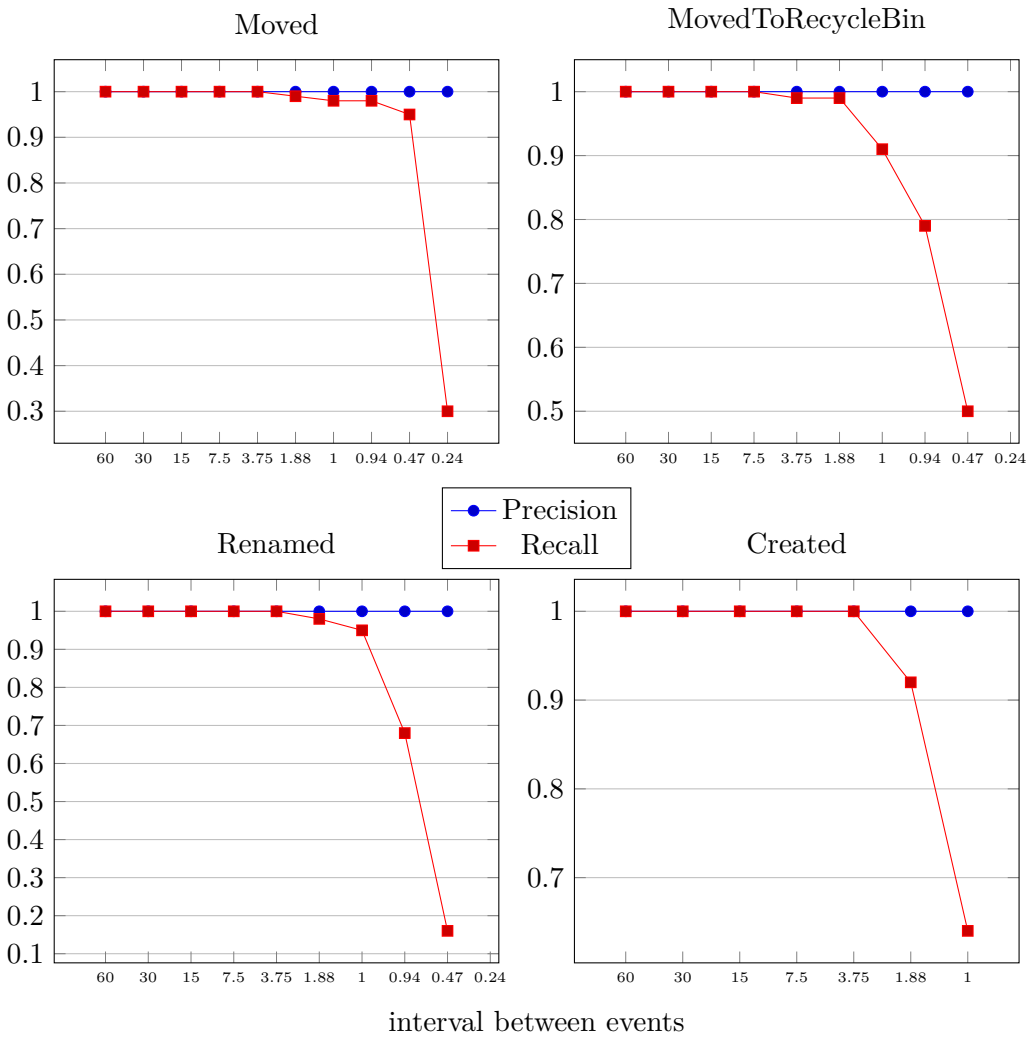


Figure 7.1: 5 minute tests results of single event types

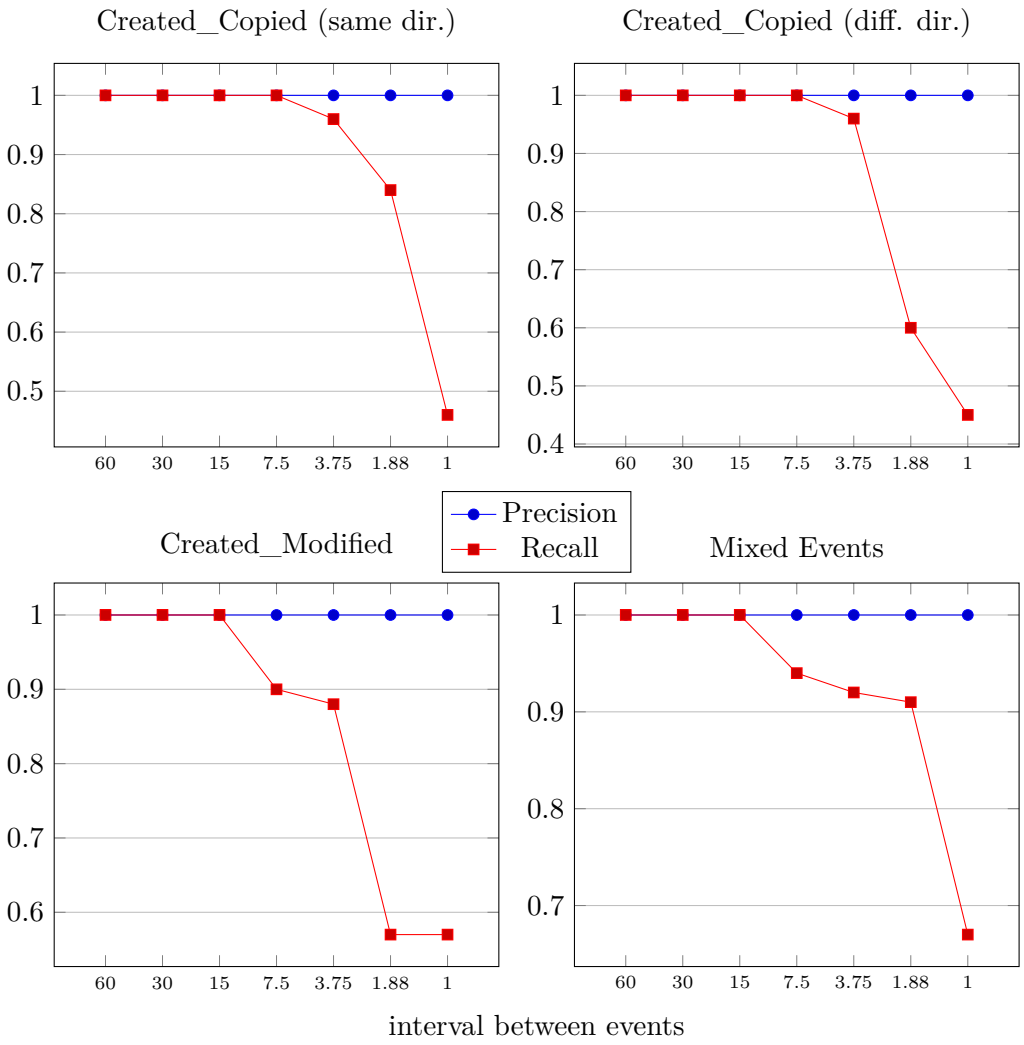


Figure 7.2: 5 minute test results of single event types and a mixed sequence of events

Time	Moved			MovedToRecycleBin			Renamed			Created		
	Logs	Precision	Recall	Logs	Precision	Recall	Logs	Precision	Recall	Logs	Precision	Recall
60	70	1	1	50	1	1	80	1	1	1 000	1	1
30	100	1	1	200	1	1	200	1	1	1 400	1	1
15	400	1	1	300	1	1	400	1	1	1 500	1	1
7.5	700	1	1	500	1	1	600	1	1	1 800	1	1
3.75	900	1	1	800	1	0.99	700	1	1	1 900	1	1
1.88	1 300	1	0.99	900	1	0.99	800	1	0.98	2 000	1	0.92
1	2 000	1	0.98	1 100	1	0.91	1 600	1	0.95	2 100	1	0.64
0.94	2 300	1	0.98	1 700	1	0.79	2 000	1	0.68	-	-	-
0.47	2 400	1	0.95	2 400	1	0.5	2 600	1	0.16	-	-	-
0.24	3 000	1	0.30	3 500	-	-	-	-	-	-	-	-

Time	Created_Modified			Created_Copied(same dir.)			Created_Copied(diff. dir.)			Mixed Events		
	Logs	Precision	Recall	Logs	Precision	Recall	Logs	Precision	Recall	Logs	Precision	Recall
60	2 000	1	1	600	1	1	800	1	1	1 700	1	1
30	3 500	1	1	700	1	1	900	1	1	2 000	1	1
15	5 000	1	1	1 700	1	1	1 400	1	1	2 700	1	1
7.5	9 000	1	0.90	1 900	1	1	1 700	1	1	3 000	1	0.94
3.75	18 600	1	0.88	2 000	1	0.96	2 100	1	0.96	10 000	1	0.92
1.88	19 000	1	0.57	2 500	1	0.84	3 000	1	0.60	11 500	1	0.91
1	20 300	1	0.57	3 000	1	0.64	5 000	1	0.45	11 600	1	0.67
0.94	-	-	-	-	-	-	-	-	-	-	-	-
0.47	-	-	-	-	-	-	-	-	-	-	-	-
0.24	-	-	-	-	-	-	-	-	-	-	-	-

Table 7.6: Results of five minute test iterations

Clients	Intervals	Precision	Recall
1	random	1	0.98
2	60-1 sec.	1	0.41
2	60-15 sec.	1	0.97

Table 7.8: 1 hour test run results with random intervals

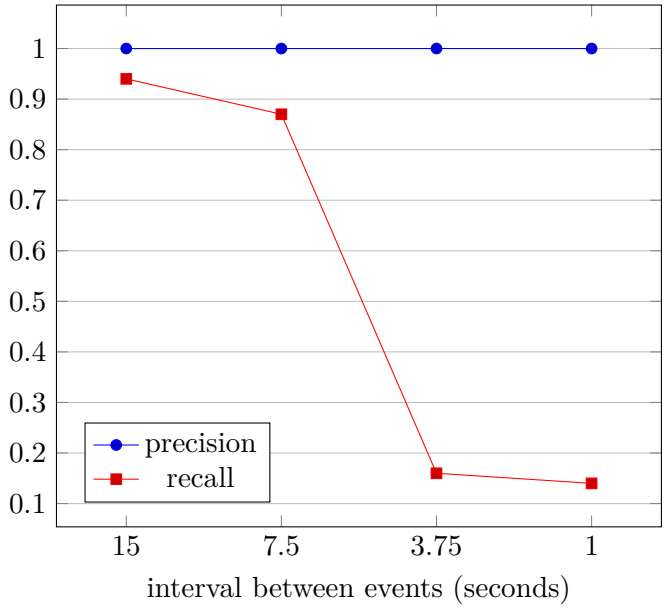
Figure 7.1 shows the decrease of the waiting time between events down to *0.47* for the types *Moved*, *MovedToRecycleBin* and *Renamed*. However, the types *Created*, *Created_Copied*, *Created_Modified* show limitations on a waiting time of *1* second. Event type *Moved* shows the best results on the smallest interval, which is *0.47* seconds. Types *MovedToRecycleBin* and *Renamed* produce similar results, whereas *MovedToRecycleBin* shows a higher decrease on detected events as *Renamed*. The *recall* of event type *Created* starts decreasing at an interval of *1.88* seconds. Copy operations detected by the type *Created_Copied* start to decrease at an interval of *3.75* seconds. However, we discover the worst results on event *Created_Modified* which already shows a lower *recall* value on an interval with *7.5* seconds and reaches its limits on an interval of *1.88* seconds. Reasons for that are the large amount of access calls triggered by programs when editing a file. We observed on past tests that a single modification activity results in multiple identical access calls and log entries. The number of log entries can be at least three times as much on a single modification, as on event type *Moved*, which produces a single log entry for one file movement. Thereby, the performance and the throughput of detected events suffers.

On a mixed sequence of events (see Figure 7.2), we discover a decrease in the *recall* value on an interval of *7.5* seconds. However, we need to consider that a higher or lower amount of events, which separately decrease or increase the throughput, can affect the overall results on mixed events. A higher number of type *Created_Modified* would decrease the *recall*, whereas a higher number of type *Moved* can result in a higher *recall*. The conducted test runs contained a balanced number of all file activities.

Load tests We executed test iterations of one hour with fixed intervals and randomly chosen waiting times between events. The third and fourth entry in Table 7.3 presents the exact setup. The goal of defined load tests is to evaluate if a longer testing period has an effect on the overall results of detected events.

In our first iterations, we used fixed intervals of *15*, *7.5*, *3.75* and *1* second. Figure 7.3 displays the results. As we can see, the *precision* is always 1 and therefore no unexpected event patterns were found. However, the *recall* is on average lower compared to shorter test iterations of five minutes. Also, the iterations with an interval of *3.75* seconds shows a significant drop of the *recall*. The waiting time of *1* second resulted in an even lower *recall*, compared to shorter iterations.

In addition, we performed an iteration using randomly chosen intervals in between file activities. Thereby, we aim to find differences in recognized events if our system does not produce events in a uniform throughput. The first entry in Table 7.8 presents the results



15		7.5		3.75		1	
Precision	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**
1	0.94	1	0.87	1	0.16	1	0.14

Figure 7.3 & Table 7.7: Scenario 3: 1 hour tests results with fixed times

of this iteration. The results do not affect the *precision*. However, according to the *recall* value, the server has not detected all events successfully. Compared to iterations using fixed waiting times, we detected a higher ratio of events.

In our last scenario, we evaluate the recognition of events receiving from two clients. The second and third entry in Table 7.8 displays the results. The executed script chooses file operations and waiting times randomly, as in the previous test run.
We performed two iterations. On the first test run we included waiting times between events from *60* seconds down to *1* second. However, the increased number of clients also raised the number of events that the system needs to process and recognize. Therefore, we discovered a low *recall* value, which indicates a low number of detected file events. On the second test run we only processed events from both clients with waiting time between *60* seconds to *15* seconds. Thereby, the activities produced fewer events that our system needs to process.

Conclusions on performed tests The following points summarize the gained knowledge and encountered limitations during the evaluation.

1. **Performance varies for different file activities:** The continuous reduction of the intervals between events resulted eventually in a clear reduction in successfully

discovered events. However, the limit varied depending on the performed file activity. This was due to the fact that each file activity produces a different sequence and amount of log entries that need to be processed. The event type *Created_Modified* produces the biggest amount of log events and therefore shows the worst performance.

2. **Limitations caused by low performance of *C-SPARQL*:** Limitations encountered during performed iterations are primarily caused by the inefficiency of *C-SPARQL* to perform continuous queries over a large amount of incoming log data. Previous works, which focused on measuring the performance of *C-SPARQL*, discovered similar results. The following listing summarizes the main issues of the engine:

   - *C-SPARQL engine crashed:* Gao et al. [2018] reports the crashing of the engine when running a query involving large static datasets and the engine reports a *ConcurrentModificationException*. The exception is caused when the process of loading data into the *Jena* database instance or clearing the data takes longer than the execution time of performing a query. We discovered the occurrence of the exception *ConcurrentModificationException* frequently on test runs with lower intervals, which produced more log data than *C-SPARQL* was able to handle. In addition, also the execution time and the memory used increases when running C-SPARQL with an increased stream rate [Ren et al., 2016]. We also encountered Java VM *OutOfMemoryExceptions* on long test runs with short intervals.
   - *High memory usage:* In case we reach a threshold, the results provided by C-SPARQL are erroneous, which Ren et al. [2016] discovered during experiments. The growth of static data also causes high memory usage of the engine [Ren et al., 2016]. C-SPARQL queries are still executed in each window regardless of currently present static data. This causes unnecessary processing overhead [Rinne et al., 2016]. Ren et al. [2016] and Gao et al. [2018] report that the reason behind those issues is that C-SPARQL does not have enough time to process both current and incoming data.
   - *Leak in-between windows:* Gao et al. [2018] also mentions that data around window borders will leak from the current window to the next window, and windows close earlier as expected. This issue would cause a file event not being detected.

3. **Query window size has to be balanced:** Choosing the correct window size can be challenging. The following points specify what we need to consider when choosing a window size:

   - *Frequency of query execution:* In the case of a smaller window size, the engine executed the query more frequently which causes frequent duplicates from the same events and produces overhead [Rinne et al., 2016]. However, on longer

window sizes the engine collects more events before the next execution and the processing time of a query execution increases. The frequency of query execution also affects the reporting time of detected events. The more time passes until the next window closes the more time it takes to detect newly incoming events and the recognition time slows down.

- *Handling of duplicate event detection:* In case we detect the same file activity repeatedly, the engine constructs identical *File Access Events*. Therefore, we require to handle duplicate events that we accomplish by filtering for the first occurrence of a detected event. This can be achieved by ordering the results by the *dateTime* field [Rinne et al., 2016].
- *Consider composite events:* When defining the window size we have to consider that composed events of multiple log records have to occur in the same window, in order for us to detect the event. One example would be a copy operation, which is a composite of two activities.

In summary, the window size affects the required resources, processing time, and notification time until the engine can report the detection of an event. Therefore, a balance of the frequency of detected events, acceptable overhead, and delay in notification time has to be found.

4. **Alternative software architecture:** In the presented architecture we use one *C-SPARQL* engine. Due to discovered performance issues of a single *C-SPARQL* engine an alternative software architecture including multiple instances of the engine could produce better results. However, exact implementation options and measurements would need to be evaluated as future work. This would include an examination if an engine for each client is feasible and if multiple engines for each query might result in a higher success rate on shorter intervals between events. Also, alternative languages over continuous queries over streams of RDF data, such as *CQELS* or *INSTANS*, could be more adequate [Ren et al., 2016] [Rinne et al., 2016].

To conclude the automated performance tests, the results show limitations in regards to detection *recall* with a high number of static event data. The previous work from Kurniawan et al. [2019b] evaluated a similar approach on Windows file logs. The results are comparable to our findings. In addition, we experienced that the number of events produced is highly dependent on the file operations, which then influences the amount of data necessary our system has to process. Furthermore, the window size influences the execution frequency and the need to handle duplicate event detection. Also, we have to consider the detection of composite events. An alternative approach regarding our software architecture may improve occurred limitations. However, exact evaluations have to be conducted as future work.

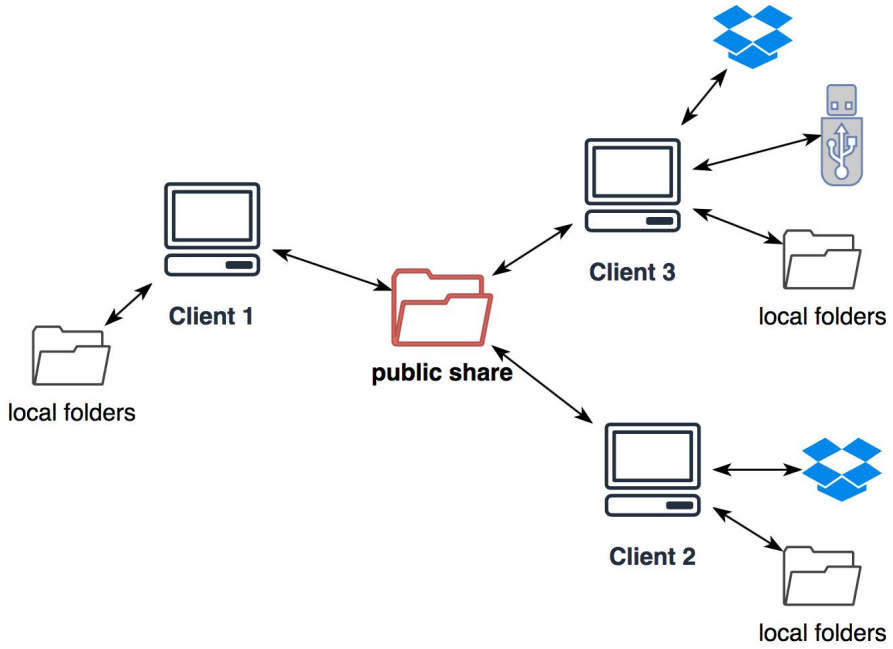


Figure 7.4: Clients setup of Data Exfiltration Scenario

7.2 Data Exfiltration Scenario

In this scenario we present user scenarios, leading to data exfiltration. To this end, we include multiple clients to produce a more realistic setup. The focus here is to include mainly event types such as *Moved* and *Created_Copied*, which exfiltrate files to an external storage such as *Dropbox* and a *USB* device.

The presented scenario should help to evaluate our second research question defined in Section 1.3, i.e. if we are able to reconstruct a file life-cycle by semantically represented log data. In addition, we ask if gathered information, by lifting file system events into RDF data, assist the identification of potential data exfiltration.

7.2.1 Experimental Setup

Our scenario includes three clients, which can access an organization's internal file share to exchange files. Figure 7.4 shows the setup of clients involved in our scenario. In addition, two clients have access to their private *Dropbox* folder and one client can also access a USB device. Thereby, we can represent a scenario in which clients are able to copy and move files from the *public share* to their local directory, a *Dropbox*, and a USB device.

Figure 7.5 presents the structure of the public share. The root folder contains a collection of four files. In addition, two subfolders are available, which contain four additional files each. File types included are *xlsx*, *docx*, *txt*, and *xml* files.

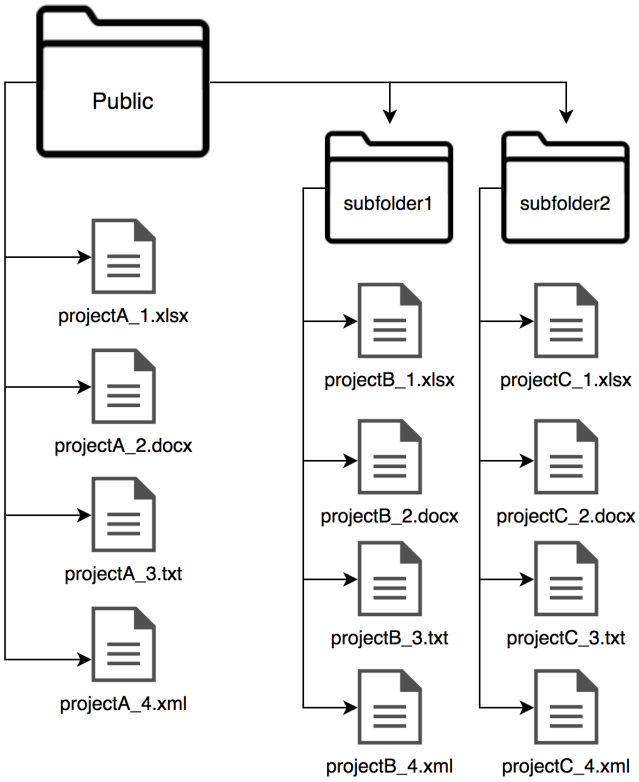


Figure 7.5: Folder structure in public share folder

In order to replicate file activities, we developed a script to perform file events. Table 7.9 describes the exact sequence of performed operations. We perform the activities of each client in sequence. In addition, we used longer intervals between events, which ensured the detection of all performed activities.

7.2.2 Evaluation Results

We constructed the file life-cycle of some interactions to visualize the performed activities. Thereby, we aim to show the ability of our prototype system regarding the reconstruction of a file history. The next section presents and describes the constructed file life-cycles. In addition, we evaluate the use of defined *Background Knowledge* and which conclusions we can draw by integrating defined knowledge of *User Accounts* and *Exfiltration Channels* to semantically represented log data. Section 5.5 describes the used models.

Graphs of File Life-Cycles This section discusses constructed file life-cycle graphs. We present one graph for each involved client, containing performed file activities.

#	**Steps client 1**
1	Client 1 copies files *projectA_3.txt* and *projectA_4.xml* from folder *Public* to a local folder */Desktop/scenario/*.
2	Client 1 renames copied files to *projectA_3_copied.txt* and *projectA_4_copied.xml*.
3	Client 1 moves files *projectA_3_copied.txt* and *projectA_4_copied.xml* from local folder back to folder *Public*.
4	Client 1 deletes original files in *Public projectA_3.txt* and *projectA_4.xml* and moves them to the recycle bin.
5	Client 2 copies file *projectA_4.xml* from folder *Public* to private *Dropbox*.

#	**Steps client 2**
1	Client 2 moves two files from *Dropbox* to folder *Public*. The files are *dropbox_file_client3_1.xlsx* and *dropbox_file_client3_2.xml*.
2	Client 2 copies the file */Public/subfolder1/projectB_3.txt* to the *USB* device.
3	Client 2 creates and edits a new local file */Desktop/scenario/local_file_client3_1.txt*.
4	Client 2 moves the new local file *local_file_client3_1.txt* to folder *Public*.

#	**Steps client 3**
1	Client 3 copies all four files from directory *Public/subfolder2/* to the local directory */Desktop/scenario/*. The files are: *projectB_1.xlsx*, *projectB_1.xlsx*, *projectB_3.txt* and *projectB_4.xml*.
2	Client 3 copied all previously copied files (Step 1) to a private *Dropbox* folder.
3	Client 3 copies file */Public/local_file_client3_1.txt* directly to a private *Dropbox* folder.
4	Client 3 renames all four previously copied files in *Dropbox* (Step 1 & 2) to *projectB_1_renamed.xlsx*, *projectB_2_renamed.docx*, *projectB_3_renamed.txt*, and *projectB_4_renamed.xml*.

Table 7.9: Steps of each client in our scenario

Figure 7.6 shows the life-cycle of file */Volumes/Public/projectA_4.xml*, produced by the first client. The first section in Table 7.9 includes the performed steps. Within the life-cycle we can see, that the user replaces file */Volumes/Public/projectA_4.xml* from the file-share by a copied version, which was modified locally. The original file got deleted.

Figure 7.7 shows a file operation included in the second section in Table 7.9, which the second client produced. The history graph visualizes a copy operation of file *projectB_3.txt* from directory *subfolder1* in the file share to path */Volumes/USB/*, which is the mount path of a USB device. Thereby, we can identify that the user copied a file to a target source that is outside of the companies' boundaries. Consequently, the user exfiltrated the file *projectB_3.txt*.

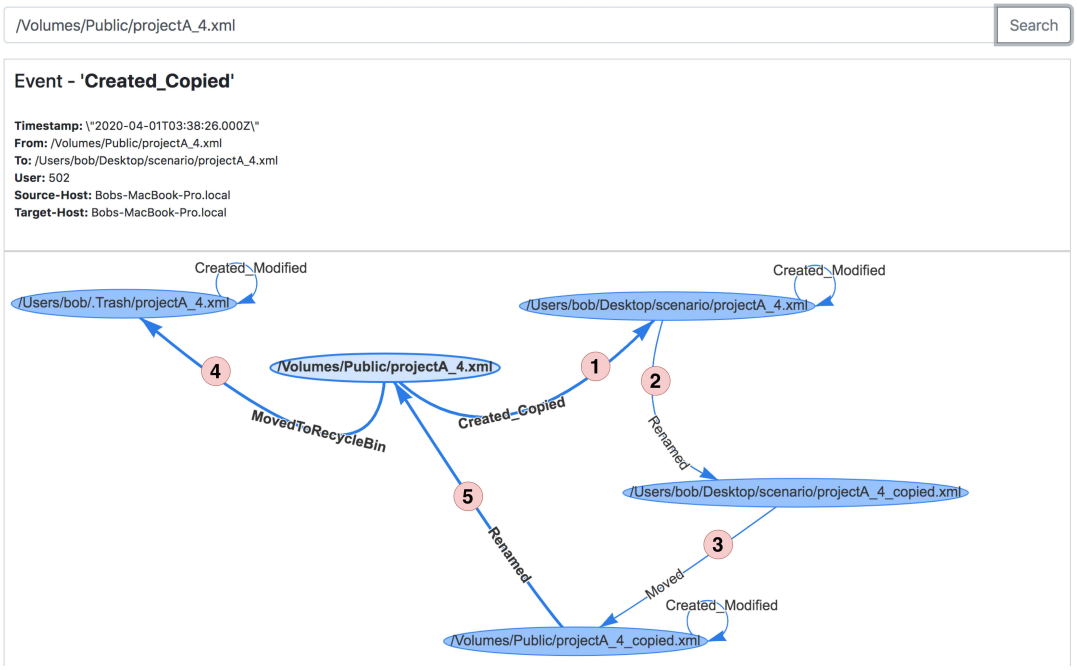


Figure 7.6: File life-cycle produced by client 1

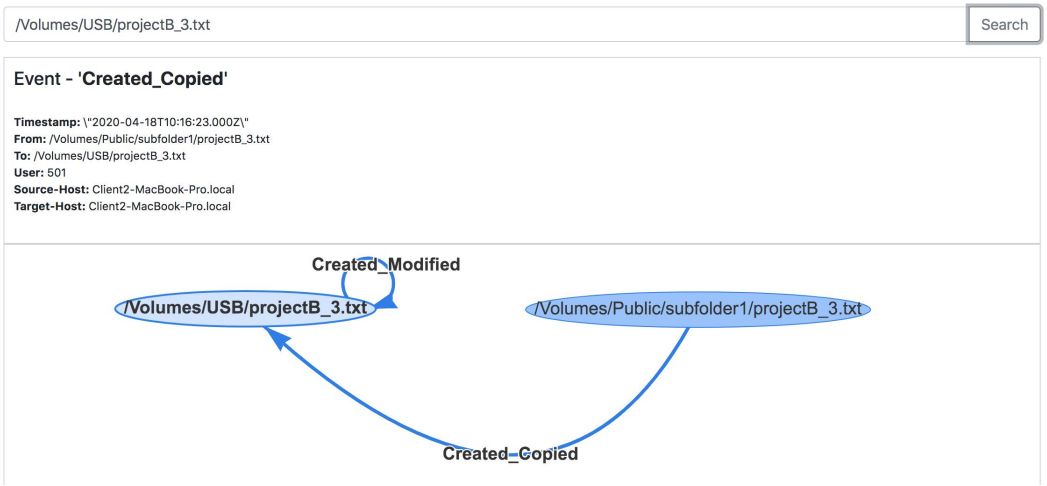


Figure 7.7: File life-cycle produced by client 2

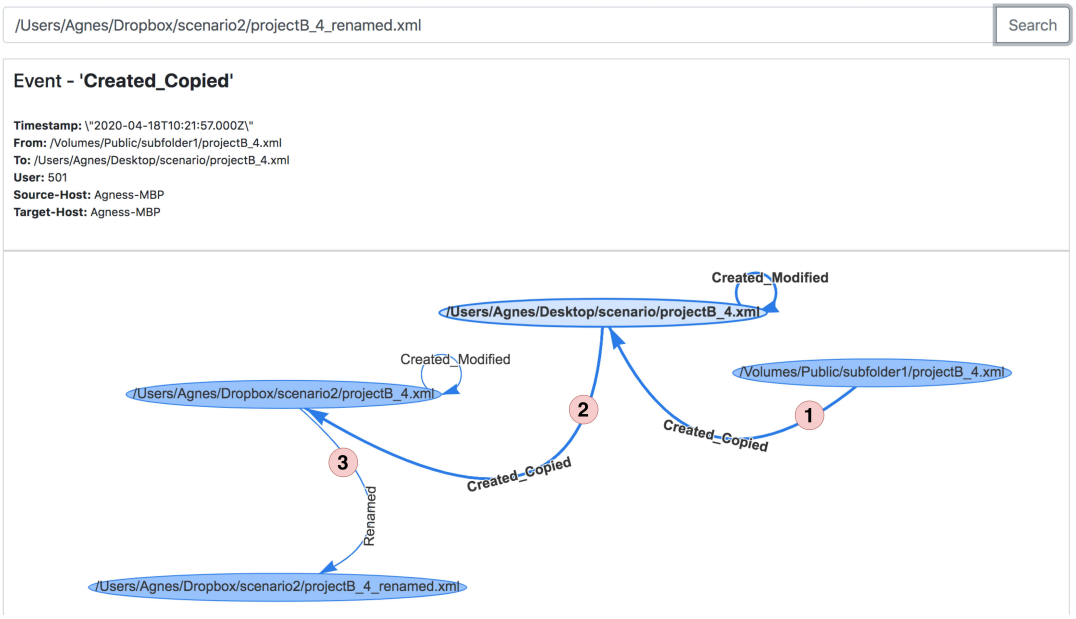


Figure 7.8: File life-cycle produced by client 3

Figure 7.8 involves file operations from the third client, which we describe in the third section in Table 7.9. As we can see in the graph, the user copied the file *projectB_4.xml* from the share to a local directory. Furthermore, the user copied the file again to a *Dropbox* folder and renamed the file. Thereby, the user exfiltrated the file to a *Dropbox* folder.

Figure 7.9 illustrates a history graph of file activities including two clients. The activities include the operations *3* and *4* in the second section in Table 7.9, performed by client 2 with host-name *Client2-MacBook-Pro.local*, and operation *3* in the third section in Table 7.9, performed by client 3 with host name *Agnes-MBP*. The file *local_file_client3_1.txt*, which the second client created and moved to the file share, was later copied to *Dropbox* by the third client.

In general, our prototype system is able to create file life-cycle graphs. Also, we are able to construct transformations and split paths. However, in order to draw conclusions about file exfiltrations and identifying responsible users, we require to integrate further knowledge into represented log data. Additional knowledge should help to reason if a move or copy operation is actually performed to an exfiltration channel. Therefore, we discuss the background knowledge in the subsequent section.

Integration of Background Knowledge Our defined *Background Knowledge* models (see Section 5.5) add more information to represented data and thereby help to draw conclusions detected activites.

In order to identify the person behind logged file activities, we integrate the *User Account*

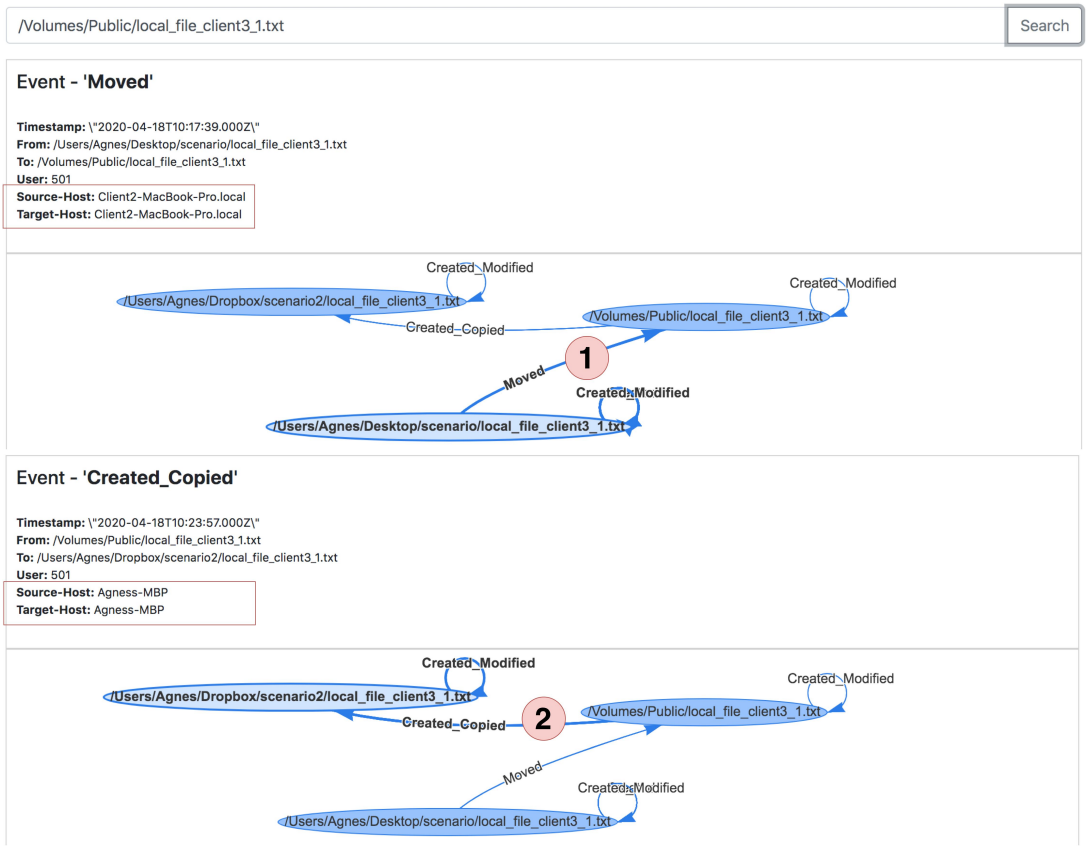


Figure 7.9: File life-cycle produced by actions of client 2 and client 3

background knowledge. This model maps the user *id* and *host name* to a user account, containing the username, name, and email address. In addition, we are able to identify file paths as *internal* or *external*, with the help of the background knowledge model *Exfiltration Channel*. Thereby, the model categorized each path and we are able to search for copy or move operations from an internal source path to an external target path. In our scenario, we consider file paths inside the file share and local directories of all clients as *internal* paths. We categorize known paths to a *Dropbox* and USB device as *external* file paths.

Listing 7.1 shows an example *SPARQL* query which uses both background knowledge models. The query searches for copy operations (line 15-16) performed by the user with username *Agnes* (line 7-8). In addition, we filter for all source paths contained in an *internal* path and all target paths, which do not contain an *internal* (lines 11-12, 18-22). Thereby, the query searches for any data exfiltrations of a specific user. Table 7.10 shows the result of the performed *SPARQL* query. The result contains six copy operations from a user. We can see that the user exfiltrated data from two clients. On one copy operation the client copied data to a USB device and on the remaining operations data was copied to a *Dropbox* folder.

```
PREFIX b: <http://w3id.org/sepses/vocab/background#>
PREFIX fae: <http://w3id.org/sepses/vocab/event/fileAccess#>
SELECT DISTINCT
?usernameBk ?pathNameSource ?pathNameTarget ?targetHostName
WHERE {
    ?bk1 b:uid ?uidBk.
    ?bk1 b:userName ?usernameBk .
    ?bk1 b:userName "Agnes" .
    ?bk1 b:host ?hostBk .

    ?bk3 b:type "intern" .
    ?bk3 b:path ?pathIntern .

    ?event      fae:timestamp                   ?timestamp .
    ?event      fae:hasAction/fae:actionName    ?actionName .
    FILTER ( ?actionName = "Created_Copied" ) .

    ?event      fae:hasSourceFile/fae:pathName  ?pathNameSource .
    FILTER ( CONTAINS( ?pathNameSource, ?pathIntern ) ) .

    ?event      fae:hasTargetFile/fae:pathName  ?pathNameTarget .
    FILTER ( !CONTAINS( ?pathNameTarget, ?pathIntern ) ) .

    ?event      fae:hasUser/fae:userName ?username .
    ?event      fae:hasTargetHost/fae:hostName  ?targetHostName .

    FILTER ( ?username =  ?uidBk ) .
    FILTER ( ?targetHostName =  ?hostBk ) .

} ORDER BY ASC(?timestamp)
```

Listing 7.1: SPARQL query to find copy operations to an external path of a user

Conclusions on performed scenario

1. **Reconstruction of file life-cycle:** The reconstruction of the file life cycle that has occurred can be successfully created provided we have discovered all events. Consequently, we require to detect all events that occurred on the respective file. In case of an incomplete discovery of events, a complete reconstruction of the life cycle is no longer possible.

2. **Background knowledge:** We integrate defined background knowledge models (presented in Section 5.5) into *SPARQL* queries. As demonstrated, with defined background knowledge we are able to filter e.g., for events performed by specific users and for events regarding *internal* and *external* file paths. However, knowledge about the file paths and the user id has to be manually defined in advance.

Field	Value
usernameBk	"Agnes"
pathNameSource	"/Volumes/Public/subfolder1/projectB_3.txt"
pathNameTarget	"/Volumes/USB/projectB_3.txt"
targetHostName	"Client2-MacBook-Pro.local"
usernameBk	"Agnes"
pathNameSource	"/Users/Agnes/Desktop/scenario/projectB_3.txt"
pathNameTarget	"/Users/Agnes/Dropbox/scenario2/projectB_3.txt"
targetHostName	"Agness-MBP"
usernameBk	"Agnes"
pathNameSource	"/Users/Agnes/Desktop/scenario/projectB_2.docx"
pathNameTarget	"/Users/Agnes/Dropbox/scenario2/projectB_2.docx"
targetHostName	"Agness-MBP"
usernameBk	"Agnes"
pathNameSource	"/Users/Agnes/Desktop/scenario/projectB_4.xml"
pathNameTarget	"/Users/Agnes/Dropbox/scenario2/projectB_4.xml"
targetHostName	"Agness-MBP"
usernameBk	"Agnes"
pathNameSource	"/Users/Agnes/Desktop/scenario/projectB_1.xlsx"
pathNameTarget	"/Users/Agnes/Dropbox/scenario2/projectB_1.xlsx"
targetHostName	"Agness-MBP"
usernameBk	"Agnes"
pathNameSource	"/Volumes/Public/local_file_client3_1.txt"
pathNameTarget	"/Users/Agnes/Dropbox/scenario2/local_file_client3_1.txt"
targetHostName	"Agness-MBP"

Table 7.10: Result of the SPARQL query from Listing 7.1

The construction of a file history graph and the integration of modeled background knowledge enables us to reason about represented log data. Vertices of the graph represent the updated filename and edges visualize the performed file activity. Thereby, we put the data into a more familiar context. Relationships between file activities and exfiltration channels become easier to understand. A representation of the file history in a tabular view is much harder to read. We can use a single *SPARQL* query to retrieve events performed over an exfiltration channel or events produced by a specific user. This enables us to potentially infer activities leading to data exfiltration. However, preconditions have to be met in order to enable further analysis. This includes the successful detection of all performed events, conducted on the analyzed file. Furthermore, involved user ids, host-names, and file paths have to be defined and integrated into the background knowledge.

7.3 Comparison with existing approaches

In the following sub-sections, we compare existing forensic tools to our implementation as well as identify differences and similarities to our approach. Furthermore, we discuss an alternative approach for semantic complex event processing to *C-SPARQL*.

7.3.1 Forensic Analysis Tools

Comparable tools include open source and commercial tools. Open source tools are: *Plaso* and *Timesketch*. Commercial tools are: *Code42*, *ADAudit Plus* and the *Log&Event Manager*. Section 4.3 describes the discussed tools in more detail. Aspects we aim to compare are tracing features of file history timelines in near-real time and graph visualization concerning file events. We describe similarities and differences of those tools to our approach in Table 7.11.

In summary, *Plaso* and *Timesketch* provide similar approaches as our solution. However, the user has to perform the analysis and search activities for suspicious activities manually. Also, the tools focus on post evaluation of digital evidence. Our approach aims for an automated examination and aggregation of collected data in near-real time. Our solution is similar to these tools with regard to filter opportunities concerning logged event attributes. Existing commercial tools are comparable to our implementation in terms of an automated solution that aims to perform analysis of occurred file activities in near real-time. However, the presented commercial tools are restricted to Windows file system events.

7.3.2 Alternatives to *C-SPARQL*

Due to encountered performance issues caused by *C-SPARQL* on a large number of static log date, we discuss different query languages as alternatives to semantic complex event processing.

As mentioned in Section 3.4, currently no standard query language for RDF streams exists. We chose *C-SPARQL* in our implementation due to the fact that it presents an extension of the popular query language SPARQL. Also, the amount of documentation and examples available were sufficient to integrate a *C-SPARQL* engine into our implementation.

Previous works [Ren et al., 2016, Rinne et al., 2016, Gao et al., 2018] already measured the performance of *C-SPARQL* compared to the languages *CQELS* and *INSTANS*. In the following sections, we present identified advantages and disadvantages of those languages compared to *C-SPARQL*. However, performance evaluations of those alternatives are beyond the scope of this thesis and are left for future work.

CQELS Based on the works of Ren et al. [2016] and Gao et al. [2018] the language *CQELS* offers features to increase the performance for querying over a large set of semantic data. One main difference between *CQELS* and *C-SPARQL* is how the languages execute queries. *CQELS* provides its own query processing framework plan and follows an eager

Tool(s)	Similarities	Differences
Plaso	• The tool can collect only events from predefined files by defining location of monitored files. • By the tool *log2timeline* a filtering for attributes within an event can be performed.	• The tool gathers information usually after a data exfiltration occurred. We focus on a near-real time approach. • It supports a much broader range of log data for analysis and not only file system logs • The analysis has to be performed manually.
Timesketch	It supports a graph integration to explore relationships between events, which is similar to our approach of reconstructing a file life-cycle.	The tool is used in conjunction with *Plaso* and is only used for the data analysis. It doesn't support the collection of log data.
Commercial tools (*Code42*, *ADAudit Plus*, and *Log&Event Manager*)	• The tools monitor file system events in order to prevent unintended file extractions outside a company's boundaries. • The tools trace file usages, shares and changes. • They aim to find misuse and theft, and create reports of reviewed activities. • Commercial tools focus on data analysis in real-time for log data examination.	Commercial tools only support Windows Server log data and file system logs of the Windows operating system.

Table 7.11: Similarities and differences to other tools

execution strategy. *C-SPARQL* uses an external system to manage windows and to process the query and executes queries periodically [Gao et al., 2018]. Performance optimizations supported by *CQELS* are adaptive generating query execution plans, caching, and encoding the intermediate result. By encoding RDF nodes as integers, the language aims to reduce the data size, and operations on integers are more efficient as on strings. One advantage of *CQELS* is that it is more efficient and robust than *C-SPARQL* [Ren et al., 2016]. *CQELS* supports a dictionary encoding technique and dynamic routing policy. Thus, the language is efficient for simple queries and is scalable with static data [Ren et al., 2016]. An increase in static data only influences *CQEL* slightly, whereas the performance of *C-SPARQL* is significantly influenced. However, *CQELS* requires more memory, due to its mentioned optimizations on query executions. Therefore, *C-SPARQL* is more memory-efficient when processing the query. Also, *C-SPARQL* supports more expressions of SPARQL 1.1 than *QCELS*, which supports fewer operations [Ren et al., 2016]. Also, *C-SPARQL* shows more correctness on query results on multi-streams. *CQELS* suffers from a serious output mismatch in the multi-stream context. This is due to the eager execution mechanism and asynchronous streams [Ren et al., 2016].

INSTANS The language *INSTANS* is compared with *C-SPARQL* by Rinne et al. [2016]. *INSTANS* differentiates from *C-SPARQL* due to its incremental query matcher. The language does not execute queries periodically. Instead, the language runs queries immediately when new data arrives. Rinne et al. [2016] compares *C-SPARQL* and *INSTANS* and mainly focues on the notification time between the languages. Also, the work discusses the handling of delicately detected events. The advantages of *INSTANS* are the ability to handle duplicate detection of events. When using *C-SPARQL*, the periodic execution with sliding windows can result in more frequent detection of the same event, whereas *INSTANS* does not face this issue. One of the main results was that *C-SPARQL* shows a higher notification time than *INSTANS*, which is related to the window-based execution. Compared to that, *INSTANS* executes queries as soon as new data is available. Therefore, the notification time is much lower than *C-SPARQL*. However, conducted tests only included *10000* static data entries. Therefore, tests between these engines with a higher amount of static data would be required to identify differences in the performance of query executions of both engines.

Alternative query languages have their advantages compared to *C-SPARL*. However, an evaluation in the same context as this work would be necessary to draw clear conclusions as to whether alternative languages can be used and whether we are able to retain all functionalities.

CHAPTER 8

Conclusions

In this chapter, we answer our research questions, discuss open issues and limitations, and motivate some future work.

8.1 Research Question Revisited

In Chapter 1.3 we defined our research questions. In this section, we give answers to those questions.

1. **Semantic representation and analysis of file system log data**
   In our first research question, we ask if semantic approaches can help the analysis process of file system log data. In order to examine this question we integrated several technologies, such as *Logstash*, *Triplewave* and *C-SPARQL*. Each technology fulfills a specific aim and by their integration we are able to represent file system log data as RDF data in near real-time. *Logstash* filters and parses raw log data. *Triplewave* transforms filtered log data into an RDF data stream. *C-SPARQL* queries for event patterns continuously over the RDF data stream and constructs *File Access Events*. Constructed *File Access Events* represent high-level events of file activities, which helps to reason about who edits, accesses, and shares a file. The transformation of raw log data into RDF data can theoretically be achieved in near real-time. However, during our evaluation, we encountered limitations with regards to the performance of *C-SPARQL* which results in erroneous query results. Thereby, based on intervals between events and the amount of raw log data produced within a specific time period a 100% success rate of detected events was not possible to accomplish. The *C-SPARQL* engine crashed frequently in case the data amount reached a threshold. This threshold varies depending on the performed file activities. We presented the detailed results of the performance tests in Section 7.1.4. In order to overcome the recognized restrictions, we require an alternative software

architecture or our system has to reconstruct events subsequently not in real-time in case we remove *C-SPARQL*. A potential alternative software architecture could include multiple instances of *C-SPARQL* engines in order to overcome performance limitations and produce better results due to parallel execution. However, we require further investigations if multiple instances of the engine are feasible and if thereby the success rate of detected events is improved. Also, alternative languages have to be considered.

2. **Construction of a file life-cycle**
   In our second research question, we ask if the information gathered by a semantic representation and the construction of a file life-cycle helps to detect potential file exfiltration. We achieve a re-construction of a files life-cycle by a *C-SPARQL* construct query which links *File Access Events*. However, if the detection of performed events is incomplete, the history might have gaps due to missing events, which makes tracing difficult and in some cases impossible. In addition, we link background knowledge containing further information about involved user accounts and exfiltration channels. An integration of defined background knowledge helps to reason about detected events. It provides information about the person who performed the file operation and by the definition of exfiltration channels, we reason about internal and external channels. Thereby, we can filter for *move* or *copy* activities performed to an external filepath. However, maintaining the background knowledge is time-consuming. An automated solution would therefore be beneficial here.

8.2 Open Issues and Limitations

In this section, we describe open issues we encountered during the implementation and evaluation process concerning the used ontologies, technologies, language, and external tools, which we categorize in the following sub-sections:

All source and target paths have to be audited:
In Section 6.4 we explained the use of *OpenBSM* in order to get a live auditing of macOS files system logs. One parameter of *auditpipe* we need to add is a list of directories that the service should audit. However, in order to receive all processed events, we require to monitor the source and target paths. Therefore, we have to add those paths to the parameter. In case a user moves a file to an unaudited path, the system does not capture the operation and therefore we do not detect the event. The *Logstash* input plugin *Auditbeat* might solve this issue. However, this plug-in currently does not support macOS as described in Section 6.5.1. If we use *Auditbeat* we would not require extracting file system logs by hand or by a custom Bash script. We described possibilities of file log data extractions in Section 6.4.

Limited to file system logs:
The developed prototype system only processes kernel file system logs which *auditd*

produces. In case a user commits data exfiltration by uploading data to a Web-interface, the corresponding log data is not included. Therefore, the implemented system is restricted to one type of log source.

Event patterns depend on the used program or process:
Depending on the program, different numbers of access calls are made. This means that we have to include all possible patterns. For example, the event *Created_Modified* triggers a different sequence of access calls depending on the used program for editing the file. In case the program *Microsoft Excel* is used for file modifications, the following sequence of events is created:

1. The operation created a temporary file with ending *.temp*, representing the file changes. Therefore, the system detects an event *Created* or *Created_Modified*.
2. The program updates the *.temp* file on write activities. Therefore, the system detects further *Created_Modified* on the temporary file.
3. After the user saves the *.xlsx* file, the program renames the *.temp* file to the original file name. Thereby, we identify an event of the type *Renamed*.

Even though one operation triggers several *File Access Events* we have to consider those events patterns to prevent misleading event detection. In the context of a modification of a *.xlsx* file we have to acknowledge additional *Renamed* and *Created_Modified* events. Furthermore, as we described in Section 6.6, copy operations performed manually in the *Finder* or via the Bash command cp produce different patterns of log data entries. Separate processes execute the activities, which trigger a non-equal sequence of access calls. In this case, we require multiple C-SPARQL construct queries, in order to detect both patterns.

Due to the fact that the system calls are highly dependent on the used program, we require extensive knowledge about available access calls. Also, the access calls for file creation and modifications are not explicit and we cannot distinguish them in most cases, which hindered a clear analysis and a clear distinction between those event types.

Limitations and performance issues of *C-SPARQL* :
Due to discovered performance issues of *C-SPARQL*, our system cannot process an extensive stream of concurrent file system log events. We encountered this issue especially when the user performs a lot of events on short intervals. In order to solve the issue, further analysis of alternative software architectures and alternative semantic query languages has to be conducted.

Limitations of the current TripleWave implementation:
We required to change the implementation of *TripleWave* in order to process received logs to RDF data correctly based on our ontology. Due to hard-coded values in the original *TripleWave* implementation, the tool produces erroneous RDF data. Thereby, the parent class was set as subject for all following triples. In case of a

sub-class the each triple was wrongly set with the same subject from the parent class. Therefore, the current TripleWave implementation only produced correct RDF data in case no sub-class is involved. We describes the required changes in Section 6.5.2.

8.3 Future Work

For future work analysis and evaluation of alternative semantic query languages over continuous semantic data is required. In addition, an alternative software architecture containing multiple instances of *C-SPARQL* may solve performance issues. However, whether an alternative architecture can be implemented in the given context, as well as potential disadvantages must be analyzed.

In addition, further log sources have to be integrated into the implemented system in order to increase the efficiency of conducted semantic analysis in order to detect possible data exfiltration event patterns. Additionally, integrating further log sources would add more knowledge for analysis. This could e.g., include logs containing metadata of connected USB devices. Information extracted from other sources should increase the completeness and accuracy to trace file life-cycles.

List of Figures

List of Tables

Listings

Bibliography

Long Cheng, Fang Liu, and Danfeng (Daphne) Yao. Enterprise data breach: causes, challenges, prevention, and future directions. *WIREs Data Mining and Knowledge Discovery*, 2017.

David Chismon, Martyn Ruks, Matteo Michelini, and Alec Waters. Detecting and deterring data exfiltration, 02 2014.

Kabul Kurniawan, Andreas Ekelhart, Elmar Kiesling, and Fajar Ekaputra. Semantic integration and monitoring of file system activity, 09 2019a.

Kabul Kurniawan, Andreas Ekelhart, Elmar Kiesling, and Fajar Ekaputra. Cross-platform file system activity monitoring and forensics – a semantic approach, 2019b.

Faheem Ullah, Matthew Edwards, Rajiv Ramdhany, Ruzanna Chitchyan, M. Ali Babar, and Awais Rashid. Data exfiltration: A review of external attack vectors and countermeasures. *Journal of Networking and Computer Applications*, October 2017.

Tore Torsteinbø. Data loss prevention systems and their weaknesses. Master's thesis, University of Agder, 2012.

Troy Hunt. Blog article of troy hunt - data breach. `https://www.troyhunt.com/the-773-million-record-collection-1-data-reach/`, 01 2019. Accessed: 2019-04-13.

Techworld. Techworld article - collection of data breach cases. `https://www.techworld.com/security/uks-most-infamous-data-breaches-3604586`, 04 2019. Accessed: 2019-04-12.

Jon Herstein. Article box security issue. `https://blog.box.com/blog/using-box-shared-links-securely`, 03 2019. Accessed: 2019-04-13.

David Thacker. Article about safety issues of google+ api. `https://www.blog.google/technology/safety-security/expediting-changes-google-plus/`, 12 2018. Accessed: 2019-04-14.

Julia Carrie. Article about facebook security issue. https://www.theguardian.com/technology/2018/sep/28/facebook-50-million-user-accounts-security-berach, 09 2018. Accessed: 2019-04-13.

Xiaokui Shu, Ke Tian, Andrew Ciambrone, and Danfeng Yao. Breaking the target: An analysis of target data breach and lessons learned. *ArXiv*, abs/1701.04940, 2017.

BBC. British airways faces record £183m fine for data breach. https://www.bbc.com/news/business-48905907, 06 2019. Accessed: 2020-02-19.

Vlad-Mihai Cotenescu and Sergiu Eftimie. Insider threat detection and mitigation techniques. *"Mircea cel Batran" Naval Academy Scientific Bulletin*, 1, 2017.

Carl Colwill. Human factors in information security: The insider threat - who can you trust these days? *Information Security Technical Report*, 14:186–196, 2010.

Areej AlHogail. Managing human factor to improve information security in organization, 5 2017.

Prof Awais Rashid, Rajiv Ramdhany, Matthew Edwards, Sarah Mukisa Kibirige, Muhammad Ali Babar, David Hutchison, and Ruzanna Chitchyan. *Detecting and Preventing Data Exfiltration*. Lancaster University, 04 2014.

Brian Carrier. *File System Forensic Analysis*. Addison Wesley, 2005.

Eva Kostrecová and Helena Bínová. Security information and event management. *Indian Journal of Research*, 2015.

Sindhu and Meshram. Digital forensic investigation tools and procedures. *I. J. Computer Network and Information Security*, pages 39–48, 04 2012.

Shweta Tripathi and BB Meshram. Digital forensic investigation on file system and database tampering. *IOSR Journal of Engineering*, 2:214–221, 02 2012.

Igor Kotenko and Andrey Chechulin. Attack modeling and security evaluation in siem systems. *International Transactions on Systems Science and Applications*, 8, 12 2012.

M. Al Fahdi, N.L. Clarke, and S.M. Furnell. Challenges to digital forensics: A survey of researchers practitioners attitudes and opinions. *2013 Information Security for South Africa*, pages 1–8, 2013.

Paulo Quintiliano, João Costa, Flavio Deus, and Rafael de Sousa Junior. Computer forensic laboratory: Aims, functionalities, hardware and software. pages 72–75, 08 2013. ISBN 9788565069090. doi: 10.5769/C2013010.

Alan R. Hevner. A three cycle view of design science research. *Scandinavian Journal of Information Systems*, Vol. 19 : Iss. 2 , Article 4, 2007.

Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Desgin science in information systems research. *MIS Quarterly Vol. 28 No. 1*, pages 75–105, 03 2004.

Vijay Vaishnavi, Bill Kuechler, Stacie Petter, and Gerard De Leoz. Design science research in information systems. *Management Information Systems Quarterly*, 28, 01 2004.

Peter Gordon. Data leakage - threats and mitigation. *SANS Institute Reading Room*, October 2007.

Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. Taylor and Francis Group, LLC, 2010.

Amadou Fall Dia, Zakia Kazi-Aoul, Aliou Boly, and Yousra Chabchoub. C-sparql extension for sampling rdf graphs streams. *Advances in Knowledge Discovery and Management*, 7, 12 2017.

Qunzhi Zhou, Yogesh Simmhan, and Viktor Prasanna. Knowledge-infused and consistent complex event processing over real-time and persistent streams. *FUTURE GENERATION COMPUTER SYSTEMS*, 10 2016.

Marc Schaaf, Stella Gatziu Grivas, Dennie Ackermann, Arne Diekmann, Arne Koschel, and Irina Astrova. *Recent Researches in Applied Information Science*.

Syed Gillani, Antoine Zimmermann, Gauthier Picard, and Frédérique Laforest. A query language for semantic complex event processing: Syntax, semantics and implementation. *Semantic Web 1*, pages 1–40, 2017.

Robin Keskisärkkä. *Towards Semancally Enabled Complex Event Processing*. PhD thesis, Linköping University, Linköping, Sweden, 2017.

Davide Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying rdf streams with c-sparql. *ACM SIGMOD Record*, 39:20–26, 09 2010a.

Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. *A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

Jean-Paul Calbimonte, Oscar Corcho, and Alasdair J. G. Gray. Enabling ontology-based access to streaming data sources. pages 96–111, 11 2010. doi: 10.1007/978-3-642-17746-0_7.

Srdjan Komazec, Davide Cerri, and Dieter Fensel. Sparkwave: Contin- uous schema-enhanced pattern matching over rdf data streams. *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, DEBS'12*, 07 2012.

Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. Ep-sparql: A unified language for event processing and stream reasoning. *Proceedings of the 20th International Conference on World Wide Web. WWW '11*, pages 635– 644, 2011.

Mikko Rinne and Esko Nuutila. Constructing event processing sys- tems of layered and heterogeneous events with sparql. *Journal on Data Semantics 6.2*, pages 57–69, 06 2016.

Pieter Bonte, Riccardo Tommasini, Filip De Turck, Femke Ongenae, and Emanuele Della Valle. C-sprite: Efficienthierarchical reasoning for rapid rdf stream processing. *13th ACM InternationalConference on DEBS*, page 103–114, 2019.

Srinivas Krishnan, Kevin Z. Snow, and Fabian Monrose. Trail of bytes: New techniques for supporting data provenance and limiting privacy breaches. *nformation Forensics and Security, IEEE Transactions on*, 7:1876–1889, 12 2012.

Jonathan Grier. Detecting data theft using stochastic forensics. *Digital Investigation*, 08 2011.

P.C. Patel and U. Singh. Detection of data theft using fuzzy inference system. pages 702–707, 02 2013. ISBN 978-1-4673-4527-9. doi: 10.1109/IAdCC.2013.6514312.

Beatriz Pérez, Julio Rubio, and Carlos Sáenz-Adán. A systematic review of provenance systems. *Knowledge and Information Systems*, 02 2018. doi: 10.1007/s10115-018-1164-3.

Adam R. Bates and K. Butler. Linux provenance modules : Secure provenance collection for the linux kernel. 2014.

Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1111–1128, Vancouver, BC, August 2017. USENIX Association. ISBN 978-1-931971-40-9. URL `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ma`.

Kelly Shortridge. What is the linux auditing system (aka auditd)? `https://capsule8.com/blog/auditd-what-is-the-linux-auditing-system/`, 2020. Accessed: 2020-09-12.

Shiqing Ma, X. Zhang, and D. Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, 2016.

Devin J. Pohly, Stephen Mclaughlin, Patrick Mcdaniel, and Kevin Butler. Hi-fi: Collecting high-fidelity whole-system provenance. In *In Proceedings of the 2012 Annual Computer Security Applications Conference, ACSAC '12*, 2012.

Frank Adelstein. Live forensics: Diagnosing your system without killing it first. *Communications of the ACM*, 49:63–66, 02 2006.

P.S. Lokhande and B.B. Meshram. Digital forensics analysis for data theft. *The International Journal of FORENSIC COMPUTER SCIENCE*, pages 29–51, 01 2015.

Timothy M. Opsitnick, Joseph M. Anguilano, and Trevor B. Tucker. Using computer forensics to investigate employee data theft. *Cybersecurity Law Strategy.*

Johan Berggren. Thinking in graphs: Exploring with timesketch. `https://medium.com/timesketch/thinking-in-graphs-exploring-with-timesketch-84b79aecd8a6`, 12 2017. Accessed: 2019-03-02.

Jan Peter Wolf. *An Ontology for Digital Forensics in IT Security Incidents.* PhD thesis, Univeryity Augsburg, 05 2013.

Mohammed Alzaabi, Andy Jones, and Thomas A. Martin. An ontology-based forensic analysis tool. *Annual ADFSL Conference on Digital Forensics, Security and Law*, 06 2013.

Spyridon Dosis, Irvin Homem, and Oliver Popov. Semantic representation and integration of digital evidence. *Procedia Computer Science*, 22:1266 – 1275, 2013.

Alfredo Cuzzocrea and Giuseppe Pirrò. A semantic-web-technology-based framework for supporting knowledge-driven digital forensics. pages 58–66, 11 2016. ISBN 978-1-4503-4267-4. doi: 10.1145/3012071.3012099.

Clóvis Eduardo de Souza Nascimento, Felipe Ferraz, Rodrigo Elia Assad, Danilo Leite, and Victor Hazin. Ontolog: Using web semantic and ontology for security log analysis. In *ICSEA 2011*, 2011.

Piyush Nimbalkar, Varish Mulwad, Nikhil Puranik, Anupam Joshi, and Timothy W. Finin. Semantic interpretation of structured log files. In *IRI 2016*, 2016.

Mark Holliday, Mark A Baker, and Rich Boakes. Grids, logs, and the resource description framework. 09 2017.

Andrew John Clark, George Mohay, and Bradley L Schatz. Rich event representation for computer forensics. 2004.

Wolfgang Klas and Michael Schrefl. *Metaclasses and Their Application*, volume 943 of *Lecture Notes in Computer Science*, pages 71–81. Springer, Berlin, Heidelberg, 06 1995.

Andreas Ekelhart, Elmar Kiesling, and Kabul Kurniawan. Taming the logs – vocabularies for semantic security analysis. *ScienceDirect*, 2018.

Agnes Fröschl. file-system-log-stream-service. `https://github.com/agnesfroeschl/file-system-log-stream-service`, 2020. Accessed: 2020-08-30.

David Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-sparql: A continuous query language for rdf data streams. *International Journal of Semantic Computing*, 04:3–25, 2010b.

Branden Gregg and Jim Mauro. *DTrace. Dynamic tracing in Oracle Solaris, Mac OSX and FreeBSD*. Oracle, 2011.

Ashish Gehani and Dawood Tariq. Spade: Support for provenance auditing in distributed environments. *IFIP International Federation for Information Processing*, pages 101–120, 2012.

James Turnbull. *The Logstash Book*. Turnbull Press, 11 2016.

Andrea Mauri, Daniele Dell'Aglio, Jean-Paul Calbimonte, and Marco Balduini. Triplewave: Spreading rdf streams on the web. *Lecture Notes in Computer Science*, 2016.

Abraham Silberschatz, Greg Gagne, and Peter B. Galvin. *Operating System Concepts*. Wiley, 2018.

Cyril Goutte and Eric Gaussier. *Advances in Information Retrieval*, volume 3408 of *Lecture Notes in Computer Science*, chapter A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation, pages 345–359. Springer, 2005.

Libo Gao, Lukasz Golab, M. Tamer Özsu, and Güneş Aluç. Stream watdiv: A streaming rdf benchmark. In *Proceedings of the International Workshop on Semantic Big Data*, SBD'18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357791. doi: 10.1145/3208352.3208355. URL `https://doi.org/10.1145/3208352.3208355`.

Xiangnan Ren, Houda Khrouf, Zakia Kazi-Aoul, Yousra Chabchoub, and Olivier Curé. On measuring performances of c-sparql and cqels. *ArXiv*, abs/1611.08269, 2016.

Mikko Rinne, Haris Abdullah, Seppo Törmä, and Esko Nuutila. Instans comparison with c-sparql on close friends. Technical report, Department of Computer Science and Engineering, Aalto University, School of Science Konemiehentie 2, Espoo, Finland, 2016.

Appendix A Precondition Configuration Files

The following listing displays the configuration file of the *audit deamon* provided by OpenBSM.

```
$ sudo cat /etc/security/audit_control
#
# $P4: //depot/projects/trustedbsd/openbsm/etc/audit_control#8 $
#
dir:/var/audit
flags:fr,fw,fa,fm,fc,fd
minfree:5
naflags:lo,aa
policy:cnt,argv,seq,path
filesz:6M
expire-after:60M
superuser-set-sflags-mask:has_authenticated,has_console_access
superuser-clear-sflags-mask:has_authenticated,has_console_access
member-set-sflags-mask:
member-clear-sflags-mask:has_authenticated
```

Listing 1: Audit configuration file

The following configuration file shows all audit classes available for *auditd*.

```
$ cat /etc/security/audit_class
#
# $P4: //depot/projects/trustedbsd/openbsm/etc/audit_class#6 $
#
0x00000000:no:invalid class
0x00000001:fr:file read
0x00000002:fw:file write
0x00000004:fa:file attribute access
0x00000008:fm:file attribute modify
0x00000010:fc:file create
0x00000020:fd:file delete
0x00000040:cl:file close
0x00000080:pc:process
0x00000100:nt:network
0x00000200:ip:ipc
0x00000400:na:non attributable
0x00000800:ad:administrative
0x00001000:lo:login_logout
0x00002000:aa:authentication and authorization
0x00004000:ap:application
```

```
0x20000000:io:ioctl
0x40000000:ex:exec
0x80000000:ot:miscellaneous
0xffffffff:all:all flags set
```

Listing 2: Audit classes of *auditd*

Appendix B Implementation of *TripleWave*

```
R2rml.prototype.transform = function(data) {
    var triples = []
    for (i = 0; i < this.tripleMaps.length; i++) {
        var tmap = this.tripleMaps[i]
        var subject = transformMap(tmap.sMap, data)
        tmap.poMaps.forEach(function(poMap) {
        var predicate = poMap.predicate.uri
        var object = transformPOMap(poMap, data)
        var key = predicate;
        var tripleObj = {};
        tripleObj["@id"] = subject;
        tripleObj[key] = object;
        triples.push(
            tripleObj
        );
        });
    }
    return triples
}
```

Listing 3: Implementation of function transform in *r2rml-js/r2rml.js*

```
\begin{lstlisting}[language=JavaScript, basicstyle=\ttfamily\small]
EnrichStream.prototype._transform = function(chunk, enc, cb) {
    var change = chunk;
    change = this.enrich(change);
    var _this = this;
    var result=[];
    var itemsProcessed = 0;
    change.forEach(function (arrayItem) {
        jsonld.expand(arrayItem, function(err, expanded) {
            result.push(expanded[0]);
            itemsProcessed++;
            if(itemsProcessed === change.length) {
                itemsProcessed = 0;
                var element = {};
                var date = new Date();
                var dateString = dateFormat(date,
                "yyyy-mm-dd'T'HH:MM:ss.l'Z'");
```

```
                element["http://www.w3.org/ns/"+
19              "prov#generatedAtTime"] = date;
                element["@id"] = "http://Triplewave-stream-"+
21              "transformation/" + dateString;
                element["@graph"] = result;
23
                _this.push(element);
25              cb();
            }
27      });
    });
29  };
```

Listing 4: Implementation of function transform in *stream/enricher.js*