With the increased sophistication of cyber attacks, organizations are under constant threat of data breaches, disruption of business processes and reputation loss. As preventive measures are not infallible, organizations have started to more closely monitor their devices and network infrastructure for malicious activity. By swift detection of an attack at an early stage, organizations can take mitigating actions limiting the impact to their organization. This detection can be done internally or is outsourced to a Security Operations Center (SOC). The SOC deploys automated detectors that monitor devices and network traffic for suspicious events, which are subsequently sent to the SOC. Here, security operators manually analyze these events, verify whether they constitute an attack and, if required, take action.

Analyzing security events is not straightforward and requires highly skilled operators. We identified three major challenges that operators face during analysis:

1. Operators need to invest time to keep up with the latest developments in attack patterns to accurately identify threats and find appropriate mitigations.
2. Operators analyze a vast number of events, which often leads to alert fatigue where operators investigate so many events it impairs their ability to correctly distinguish malicious behavior from falsely flagged events.
3. Operators require sufficient contextual information to assess security events.

This work aims to better understand security events and applies that knowledge to develop approaches that assist (semi-)automated analysis. Concretely, we first investigate the process of sharing threat intelligence through reports describing high-level tactics and techniques used by attackers. In doing so, we develop a natural language processing framework that automatically extracts actionable threat intelligence and classifies it into the ATT&CK knowledge base, a framework describing threat models and methodologies. Second, we study the event investigation process known as triaging. Here, we develop an approach that semi-automatically analyses security events in the context of other security events to determine the overall risk level. Third, we deeper investigate security events on the network level and devise an approach that clusters encrypted network traffic according to the application that produced it. This allows security operators a deeper understanding of network traffic and allows them to more effectively block malicious activity. Finally, we perform a case study where we apply the methods developed in this work to the domain of identity and access management policies to identify misconfigurations. This case study demonstrates the potential for our methods in future work.

Combining these findings, we conclude that these approaches bring us a step closer to understanding security events and providing adequate responses.

Comprehending Security Events: Context-Based Identification and Explanation

Thijs Sebastiaan van Ede

# Comprehending Security Events

## Context-Based Identification and Explanation

Thijs Sebastiaan van Ede

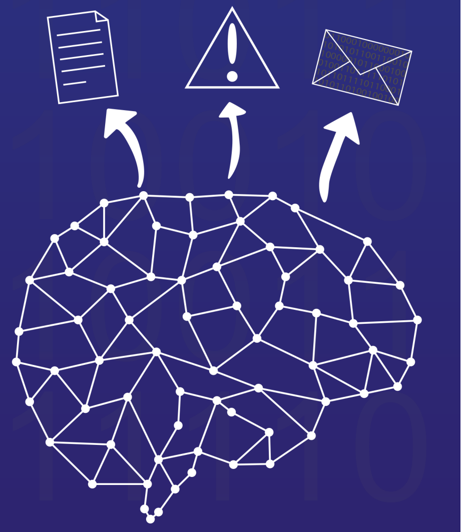
**INVITATION**

to attend the public defence of the PhD thesis:

Comprehending Security Events Context-Based Identification and Explanation

by Thijs van Ede

24 November 2023, 12:45
Waaier 4
University of Twente



# Comprehending Security Events: Context-based Identification and Explanation

Thijs Sebastiaan van Ede

# COMPREHENDING SECURITY EVENTS: CONTEXT-BASED IDENTIFICATION AND EXPLANATION

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof. dr. ir. A. Veldkamp,
on account of the decision of the Doctorate Board,
to be publicly defended
on Friday 24th of November 2023 at 12:45 hours

by

**Thijs Sebastiaan van Ede**

born on the 3rd of November, 1994
in Harderwijk, the Netherlands

This dissertation has been approved by:

Promotors:
prof. dr. M.R. van Steen
prof. dr. A. Peter

Co-promotor:
dr. A. Continella

UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE
Semantics, Cybersecurity & Services group
P.O. Box 217, 7500 AE
Enschede, the Netherlands

Digital Society Institute
P.O. Box 217, 7500 AE
Enschede, the Netherlands

Cover design: Joris van Ede
Printed by: Ipskamp Printing
Lay-out: Thijs van Ede
ISBN: 978-90-365-5888-4
DOI: 10.3990/1.9789036558891

## GRADUATION COMMITTEE:

| | | |
|---|---|---|
| Chair/secretary | prof. dr. J.N. Kok | Universiteit Twente |
| Promotors | prof. dr. ir. M.R. van Steen | Universiteit Twente |
| | prof. dr. A. Peter | Universität Oldenburg |
| Co-promotor | dr. ir. A. Continella | Universiteit Twente |
| Committee Members | prof. dr. ir. R.M. van Rijswijk - Deij | Universiteit Twente |
| | prof. dr. M. Huisman | Universiteit Twente |
| | prof. dr. H. Bos | VU Amsterdam |
| | prof. dr. S. Zanero | Politecnico di Milano |
| | dr. M. Caselli | Siemens AG |

# Abstract

With the increased sophistication of cyber attacks, organizations are under constant threat of data breaches, disruption of business processes and reputation loss. As preventive measures are not infallible, organizations have started to more closely monitor their devices and network infrastructure for malicious activity. By swift detection of an attack at an early stage, organizations can take mitigating actions limiting the impact to their organization. This detection can be done internally or is outsourced to a Security Operations Center (SOC). The SOC deploys automated detectors that monitor devices and network traffic for suspicious events, which are subsequently sent to the SOC. Here, security operators manually analyze these events, verify whether they constitute an attack and, if required, take action.

Analyzing security events is not straightforward and requires highly skilled operators. We identified three major challenges that operators face during analysis:

1. Operators need to invest time to keep up with the latest developments in attack patterns to accurately identify threats and find appropriate mitigations.
2. Operators analyze a vast number of events, which often leads to alert fatigue where operators investigate so many events it impairs their ability to correctly distinguish malicious behavior from falsely flagged events.
3. Operators require sufficient contextual information to assess security events.

This work aims to better understand security events and applies that knowledge to develop approaches that assist (semi-)automated analysis. Concretely, we first investigate the process of sharing threat intelligence through reports describing high-level tactics and techniques used by attackers. In doing so, we develop a natural language processing framework that automatically extracts actionable threat intelligence and classifies it into the ATT&CK knowledge base, a framework describing threat models and methodologies. Second, we study the event investigation process known as triaging. Here, we develop an approach that semi-automatically analyses security events in the context of other security events to determine the overall risk level. Third, we deeper investigate security events on the network level and devise an approach that clusters encrypted network traffic according to the application that produced it. This allows security operators a deeper understanding of network traffic and allows them to more effectively block malicious activity. Finally, we perform a case study where we apply the methods developed in this work to the domain of identity and access management policies to identify misconfigurations. This case study demonstrates the potential for our methods in future work.

Combining these findings, we conclude that these approaches bring us a step closer to understanding security events and providing adequate responses.

# Samenvatting

De hedendaagse cyberaanvallen vormen een constante bedreiging voor organisaties en kunnen leiden tot datalekken, procesverstoring en verlies van reputatie. Omdat preventieve oplossingen niet waterdicht zijn, controleren bedrijven hun IT-infrastructuur steeds nauwkeuriger op kwaadaardige activiteiten. Wanneer aanvallen sneller worden opgespoord kunnen organisaties sneller ingrijpen wat de schade beperkt. Deze controle wordt intern gedaan of uitbesteed aan een Security Operations Center (SOC). Dit SOC plaatst detectoren die apparaten en netwerkverkeer controleren op verdachte activiteit en vervolgens doorspelen aan het SOC. De activiteit wordt handmatig door security-analisten geanalyseerd om te bepalen of er sprake is van een aanval. Indien nodig wordt hierop actie ondernomen.

Het onderzoeken van deze activiteit is niet eenvoudig en vereist bekwame analisten. In dit werk hebben wij drie uitdagingen voor analisten geïdentificeerd:

1. Analisten moeten tijd investeren om op de hoogte te blijven van de laatste ontwikkelingen in aanvalstechnieken. Hierdoor kunnen zij dreigingen nauwkeuriger identificeren en schade beter beperken.
2. Analisten moeten grote hoeveelheden aan activiteiten onderzoeken, wat kan leiden tot analyse-vermoeidheid. Deze vermoeidheid vermindert het vermogen om kwaadaardig gedrag te herkennen.
3. Analisten vereisen toegang tot voldoende contextuele data om verdachte activiteiten te kunnen beoordelen.

Dit proefschrift streeft ernaar verdachte activiteiten beter te begrijpen en past die kennis toe op het ontwikkelen van methodes die (semi)automatische analyse van deze activiteiten verbeteren. Meer specifiek onderzoeken wij eerst het proces waar analisten documenten delen die dreigingsinformatie over gebruikte kwaadaardige technieken bevatten. In dit onderzoek ontwikkelen wij een raamwerk voor het verwerken van documenten geschreven in security-specifieke natuurlijke taal. Dit raamwerk vindt automatisch dreigingsinformatie en classificeert deze in de ATT&CK databank die informatie bevat over dreigingen en gebruikte kwaadaardige technieken. Ten tweede bestuderen wij de initiële analyse van activiteiten in een SOC, wat bekend staat als triage. Hiervoor ontwikkelen wij een methode voor het semiautomatisch analyseren van security activiteiten in de grotere context van ander geobserveerd gedrag. Deze analyse stelt ons in staat het risico van kwaadaardige activiteiten sneller en nauwkeuriger in te schatten. Ons derde onderzoek duikt dieper in het bieden van extra context voor applicaties die over versleuteld netwerkverkeer communiceren. We ontwikkelen een methode die versleuteld netwerkverkeer per applicatie groepeert. Dit biedt een hulpmiddel voor security-analisten om

kwaadaardige activiteiten effectiever op te sporen en tegen te gaan. Ten slotte onderzoeken wij een casus waarin de methodes van ons onderzoek worden toegepast op gebruikersbeheer en toegangscontrole voor het detecteren van misconfiguraties. Deze casus toont de inzetbaarheid van ons werk in toekomstig onderzoek.

Tezamen laten onze bevindingen zien dat de ontwikkelde methodes ons meer inzicht geven in security activiteiten en bijdragen aan het adequaat handelen van analisten.

# Contents

# Chapter 1

# Introduction

Over the past decades, cyber attacks have become increasingly prevalent. To mitigate their impact, industry and academia have developed a plethora of detection solutions. These solutions allow security operators to quickly and efficiently identify various threats. However, existing approaches often focus on individual threats that are commonly part of a larger attack. This focus on detecting individual steps in what is known as a "kill chain" makes it difficult for operators to assess the full impact of an attack and determine and prioritize mitigation approaches. In this work, we introduce a context-based approach to analyze cyber security events that allows us to more reliably identify attacks on IT infrastructure. Besides improved automated detection capabilities, context allows us to better explain detections and decisions made by these automated systems to security operators.

## 1.1 Security Monitoring

To mitigate threats in enterprise infrastructures, organizations started to monitor their machines and networks for suspicious behavior. Large organizations with specialized security teams perform this monitoring in-house. However, with the increased complexity of attacks, specialized companies started to offer monitoring services. In either case, the (part of the) organization performing the monitoring is called the Security Operations Center (SOC), see Figure 1.1. To perform their tasks, the SOC collects *data* (e.g., system logs and network traffic) detailing device and network activity. Subsequently, this data is fed through an Intrusion Detection System (IDS) that analyzes various data sources, and flags suspicious activity as a *security event*. This automated analysis can be based on given rules, heuristics or data-driven algorithms. While these security events are essential in finding threats, they may trigger for suspicious, but ultimately benign behavior as well. Therefore, to create accurate detection approaches and assess the security events that they produce, security operators collect Cyber Threat Intelligence (CTI). This CTI describes threat behavior in the form of either Tactics, Techniques and Procedures (TTP) or Indicators of Compromise (IoCs). Where TTP describes the high-level behavior of threat actors, IoCs specify malicious indicators such as IP addresses, URLs or file hashes. IoCs can often be directly translated into detection rules, whereas TTPs are often used to prioritize and assess produced security events.

In an ideal world, all suspicious logs that are flagged as security events are malicious and appropriate action can be taken. However, suspicious events are not

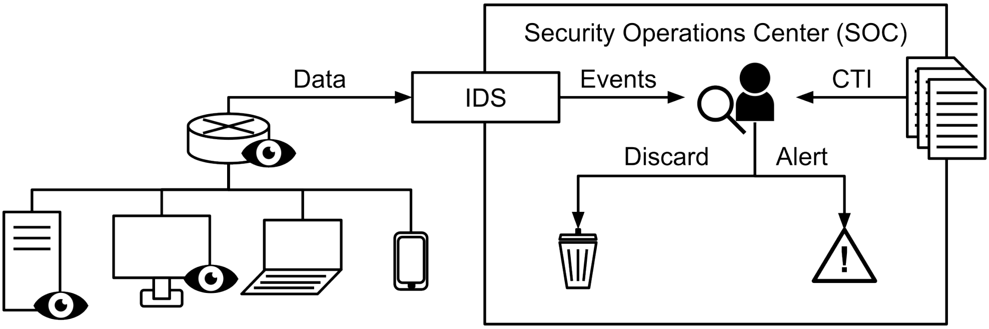
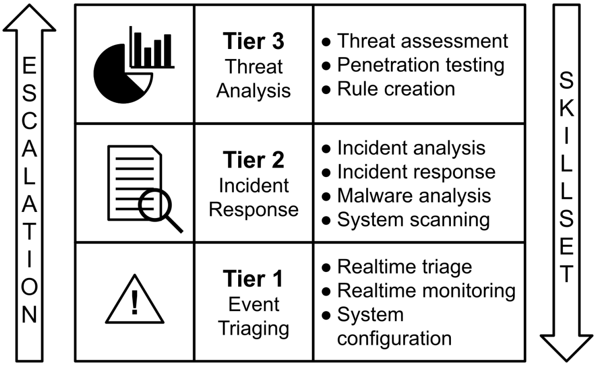


Figure 1.1: **Simplified overview of Security Operations Center.**

necessarily malicious. To illustrate, assume an event was triggered due to beaconing, i.e., periodic network activity. This beaconing activity might be triggered by a malware sample that is trying to connect to its command and control server. However, periodic activity can also be caused by a benign application trying to poll for updates. Hence, while beaconing is definitely a suspicious security event, it is not necessarily malicious. Therefore, security events are often checked by a human who decides whether a security event constitutes an alert and therefore requires action, or whether it can be discarded as being benign. This initial discerning between malicious and benign security events is called *triaging* [20, 21, 99] and is a labor-intensive and error-prone process. Moreover, the vast amount of events that must be triaged can lead to a condition called *alert fatigue*, where security operators are shown so many (benign) security events that it reduces their ability to correctly identify malicious events. This work is an effort to better understand both the security events and their triaging process and provide models that can assist in automation, explanation of decisions and performance improvements. However, to understand these steps, we first provide a background of the current workflow of SOCs, and explore some of the work that has been done in the automation of security monitoring and identify open problems therein.

### 1.1.1 Intrusion Detection

To monitor IT infrastructures for threats, organizations use IDSs, which can analyze activity on individual hosts (HIDS) or in the network (NIDS) also sometimes known as a network security monitor (NSM). These IDS monitor data that are produced by system activity via rules [85, 179], heuristics [5, 7], or data-driven (anomaly detection) models [76, 140]. In general, rule- and heuristic-based systems are widely adopted by the industry due to their extendable setup and easy interpretability by security operators. Conversely, academia focuses more on data-driven models as they are potentially able to detect treats that have not been manually defined. As



| | | |
|---|---|---|
| | **Tier 3**<br>Threat Analysis | • Threat assessment<br>• Penetration testing<br>• Rule creation |
| | **Tier 2**<br>Incident Response | • Incident analysis<br>• Incident response<br>• Malware analysis<br>• System scanning |
| | **Tier 1**<br>Event Triaging | • Realtime triage<br>• Realtime monitoring<br>• System configuration |

Figure 1.2: **SOC Tier Structure.** Common division of analysis tasks of security events based on operator tiers (image adapted from [118]).

the behavior of data-driven models is less well understood, this is an active research area. In practice, however, monitoring setups often use a combination of detection techniques to cover a large variety of threats. As it is detrimental to the security of an organization if threats are missed, these IDSs often try to detect all security events, meaning that they sometimes incorrectly detect benign behavior as well, i.e., produce false positives. Therefore, security events produced by an IDS cannot be taken as ground truth and should be analyzed by operators in a SOC before appropriate action is taken.

### 1.1.2 Security Operations Center

Upon detection by an IDS, security events are sent to a central SOC for further analysis. Due to the vast number of (false) security events that a SOC receives, the first step is to prioritize events and filter out false positives. We recall that this process of assessing the maliciousness of security events is known as alert triaging.

However, alert triaging is not the only action taken by a SOC. Due to the vast number of security events, SOCs often work with operators in different tiers that perform either triaging (Tier 1), incident response (Tier 2), or threat analysis (Tier 3) [118] as shown in Figure 1.2. Security event triaging is the first step in this process and aims to quickly filter false positives from actual malicious events. Discarded events are ignored, whereas malicious events are escalated and treated as security incidents or alerts. Next, an incident responder analyzes the alert to fully understand what happened and contacts the affected organization to carry out an appropriate response. Whenever large or unknown threats are detected, a threat analyst attempts to coordinate an appropriate response and may update the IDS with new rules, heuristics or models to better detect the threat.

As alert triaging is the first step, it sees the largest number of alerts and is therefore the most time-consuming task within a SOC. Due to the high workload on operators, triaging often leads to a condition called *alert fatigue*, where security operators fail to respond to alerts because of the sheer volume they receive each day [99]. Therefore, academic literature focuses on improving the efficiency of Tier 1 security operators by improving the precision of IDS detectors; prioritizing security events based on the type of detected event; and automating the workflow that operators take when triaging events. In this latter workflow, operators focus not only on individual security events but analyze them in the context of other events.

### 1.1.3 Contextual Analysis

Where IDSs detect individual security events, a larger attack often consists of multiple steps that can each trigger multiple events. The collective steps of an attack are known as a *kill chain* and can be used by security operators to improve their triaging by correlating events. Furthermore, once a kill chain has been identified, the security operators can better predict the possible next steps of an attacker and improve their incident response. Kill chains can be described in terms of Tactics, Techniques and Procedures (TTPs) and are described by various frameworks. The MITRE ATT&CK framework [199] has become the de facto standard for describing goals (tactics) and techniques by which they are accomplished, as well as providing additional information such as tools that are used and various APTs that are known to carry out these attacks. Figure 1.3 gives a simplified overview of entities and relations described by the MITRE ATT&CK framework. While such frameworks help security operators in their analyses of security events, the process of translating contextual knowledge and correlating it with observed events remains a manual effort.

### 1.1.4 Cyber Threat Intelligence

Where contextual frameworks can assist in the analysis of security events, this knowledge must be shared among experts to provide effective countermeasures. Low-level intelligence such as IoCs are often shared through platforms such as MISP [167]. However, as our goal is to improve the workflow of SOC operators, our focus is on high-level CTI that describes TTPs. This high-level kill chain information is often shared through security reports, blog posts and white papers collectively known as CTI reports. These reports may include IoCs, but focus more on the kill chains of individual attacks and provide additional contextual information about the adversarial groups, tools and techniques that these adversaries use to obtain certain tactics. By exchanging these CTI reports security operators can share knowledge of attacks they observed in their own environments. Additionally, by reading other reports, operators can learn to better recognize attack patterns

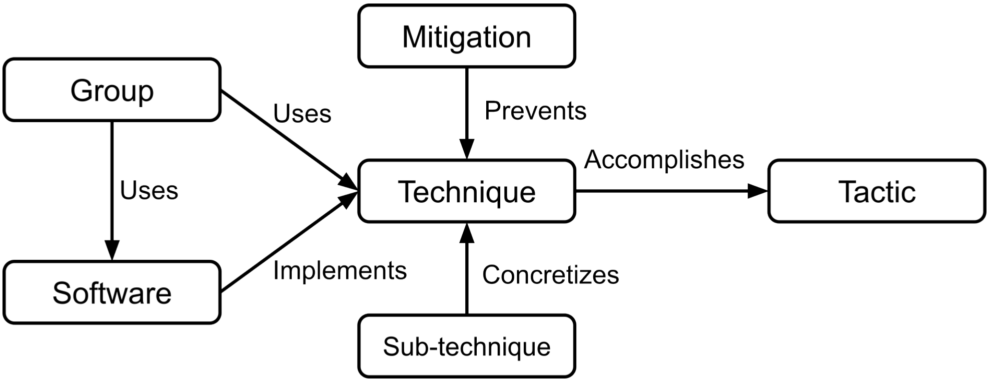


Figure 1.3: **Simplified structure of MITRE ATT&CK framework.** The MITRE ATT&CK framework defines Tactics, Techniques and Procedures (TTPs) as well as their relations, that have been observed in the wild.

seen by other experts, and potentially create additional detectors to identify these attacks. These CTI reports are often written in highly technical, natural language that is understood by experts, but difficult to interpret by laymen and machines. In an effort to make CTI reports more searchable, and to explicitly define the relations between different parts of an attack, the threat intelligence community created the Structured Threat Information Expression (STIX) language and serialization format to exchange CTI [155]. STIX allows users to express CTI in the form of JSON objects that have specified relations with other objects and can be tagged with metadata. STIX can be used to tag CTI reports with e.g., concepts from the MITRE ATT&CK framework, to allow security operators to quickly identify relevant reports. In an ideal world, all CTI reports are shared in combination with metadata tagging the report with relevant concepts from frameworks such as MITRE's ATT&CK. However, this requires additional effort from already overloaded security operators, who would have to manually tag reports. Therefore, CTI reports often do not include explicit information about the MITRE ATT&CK concepts they cover.

## 1.2 Open problems & Research Questions

The goal of this work is to better understand security events and the way they are triaged to improve the detection capabilities of a SOC. We aim to integrate contextual kill chain information in the analysis of security events. To this end, we introduce novel techniques that aim to improve the workflow of a SOC in terms of information sharing, automated event analysis, detection of security events as well as supporting preventive approaches. Together, these contributions aim to answer our main research question:

**Main RQ.** *To what extent can we leverage high-level contextual knowledge of cyber kill chains into security event analysis?*

To address this research question, we focus specifically on three different open problems and perform one case study in an adjacent domain. First, to understand security events in a broader context, security operators share knowledge about known attacks. We recall that CTI reports are written in natural language, making them easy to understand for humans with the appropriate technical knowledge. However, this same natural language makes it difficult to automatically scan, classify and extract structured threat information from such reports. Therefore, to assist security operators in searching and identifying relevant reports, we introduce a natural language processing (NLP) framework that specializes in the extraction of cyber-specific information from CTI reports. In doing so, we aim to answer the following research question:

**RQ 1.** *To what extent can we automate knowledge extraction from CTI reports and classify it into existing TTP frameworks?*

While CTI reports contain valuable information for detecting threats and triaging events, attacks that are not described in reports may be missed. Therefore, to fully understand security events in the context of an attack, and thereby assist security operators in event triaging, we propose an approach for contextual analysis of security events. Until now, such contextual analyses are either done manually, leading to alert fatigue, or are implemented as static rules [181] making it difficult to cover all possible attack chains. Instead of imposing rules as a top-down approach, we explore the possibility of bottom-up approaches, infering kill chain patterns from observed security events. By automatically correlating security events and proposing a human-in-the-loop approach for triaging groups of similarly correlated events, we try to answer the following research question:

**RQ 2.** *To what extent can we reduce the manual workload of triaging security events based on contextual information?*

While providing a workload reduction for security operators might increase their efficiency, a main assumption is that there is enough contextual information available to assess given security events. However, most IDS detectors focus on producing events for either known threats (using rules and signatures) or anomalies (using heuristics and anomaly detectors). Events produced by rules and signatures are inherently linked to specific malicious behavior. While they are relatively easy to assess in a SOC, they do not cover all malicious behavior. Conversely, anomaly detectors can identify unknown behavior and trigger security events for them. However, anomalies are not necessarily malicious and malicious behavior may not even be anomalous. Therefore, security events produced by anomaly detectors are much more difficult to assess. In the case of host-based anomaly detection, one can collect additional information through an Endpoint Detection and Response (EDR) solution. However, with the advent of bring-your-own-device policies security operators lost access to a large portion of devices within their infrastructure. Therefore, they have to resort to detecting suspicious behavior from the network, which does not provide detailed contextual information. To overcome this limitation, we investigate the following research question:

> **RQ 3.** *To what extent can we identify malicious applications based on network traffic?*

Finally, we observe that the open problems and research questions focus on faster, more accurate detection of malicious activity. However, our developed contextual identification and explanation approaches could potentially benefit adjacent domains as well. To kick off potential future research directions we perform a case study of context-based analysis in Identity and Access Management (IAM) solutions. These IAM solutions provide organizations with tools to set policies that restrict unauthorized access while still allowing designated roles within an organization to carry out their tasks. As IAM policies control access, they are essential when reducing the potential attack surface and can lead to disastrous consequences when policies are misconfigured. Due to the high level of knowledge required for designing these policies, checking if policies are misconfigured is either a highly manual approach or can only be done reactively, after an incident occurred. Therefore, in our final case study, we explore to what extent context-based analysis can assist in preventing misconfigurations in IAM Policies.

## 1.3 Contributions & Thesis Overview

Figure 1.4 provides an overview of the research areas discussed in this thesis. We begin with the investigation of cyber threat intelligence (CTI) in Chapter 2. Here, we focus on automating the analysis of CTI reports with the help of natural language processing (NLP) and extracting knowledge that helps us classify described attacks in the MITRE ATT&CK framework. To this end, we propose a framework called “EAGLE” that addresses the main challenges of NLP in the domain of security and provides a plug-and-play solution for the automated processing of CTI reports. This approach assists security event detection by providing a context-based explanation for observed events.

Next, in Chapter 3 we approach security event analysis from the contextual viewpoint of a triaging security operator. Here, we introduce DeepCASE, a semi-supervised deep learning approach, that identifies malicious security events in rela-

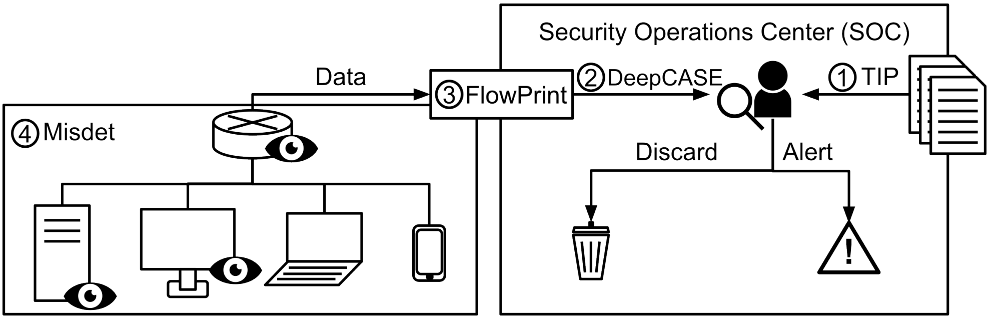


Figure 1.4: **Thesis Overview.** This thesis covers 1) Threat Intelligence Processing, our approach to automatically analyze CTI reports and extract actionable threat intelligence. 2) DeepCASE, which learns attack patterns by performing contextual analysis of security events. 3) FlowPrint, which zooms in on generating security events for new, previously unseen applications. 4) The Misdet case study applies our contextual approach as a preventive measure by analyzing security policies.

tion to other, contextual events. By automatically discovering correlations between security events, we provide an explainable approach that vastly reduces the workload of security operators.

While contextual detection vastly improves the workflow of security operators, the fundamental assumption that SOCs work on is that IDSs produce security events for suspicious detected behavior. Chapter 4 introduces FlowPrint, our approach that attempts to detect not only suspicious behavior but raises events for individual applications that are visible from the network. By identifying traffic on an application level, we provide better contextual insights to security operators when detecting threats. Moreover, because of our semi-supervised approach to detection, we cover both known and previously unseen applications, making this technique extra powerful in the context of attacks, where not all applications are known in advance.

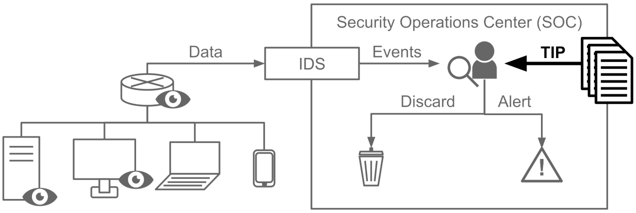
After this in-depth analysis of security events from various angles, we shift our focus from the detection of events and explore a case study in which we use a context-based approach outside of the detection domain. In Chapter 5, we leverage the contextual knowledge of an IT infrastructure to identify misconfigurations in IAM Policies. By doing so, we show the potential of context-based solutions in preventing incorrect access to resources within cloud environments (in our case AWS), and thereby reduce the attack surface of adversaries.

Finally, in Chapter 6, we reflect on the approaches introduced in this thesis, summarizing the results and discussing future research directions.

# Chapter 2

## Threat Intelligence Processing - Unleashing the Real Power of Natural Language Processing for Cyber Threat Intelligence

We start by investigating cyber threat intelligence (CTI) reports and the way we can automatically extract information that is relevant for security operators. To this end, we answer the following research question:



**RQ 1.** *To what extent can we automate knowledge extraction from CTI reports and classify it into existing TTP frameworks?*

CTI plays a critical role in sharing knowledge about new and evolving threats among security experts. However, with the increased prevalence and sophistication of threat actors, intelligence has expanded from simple indicators of compromise to extensive reports describing high-level attack steps. This higher-level view makes CTI significantly more valuable, but also harder to interpret and process. Recent advances in Natural Language Processing (NLP) have made it possible to automate large parts of knowledge extraction from these CTI reports. While several works have proposed the adoption of a variety of NLP techniques, applications have focused on highly specialized techniques, leading to insufficient, unexplainable, and often unreproducible results. In this chapter, we systematize recent advances in NLP to understand the problems they tackle and use these insights to review and contextualize existing research on the application of NLP for cyber threat intelligence. Moreover, we propose a unified and customizable plug-and-play framework EAGLE, that enables both researchers and industry to combine and apply well-known NLP techniques when processing threat intelligence. We evaluate the contribution of NLP techniques included in our framework on the task of extracting MITRE ATT&CK techniques from CTI reports and identify current limitations and future research directions.

This chapter is based on the publication: [4] Thijs van Ede et al. "Threat Intelligence Processing - Unleashing the Real Power of Natural Language Processing for Cyber Threat Intelligence". In: *(Under submission)*. 2023

## 2.1 Introduction

In our ever-evolving threat landscape, security specialists try to keep up with changing threats. To do so, they require cyber threat intelligence (CTI) to inform them of malware developments and methods to detect and mitigate threats. CTI comes in different forms: technical metadata known as indicators of compromise (IoCs), which include malicious file names and hashes, URLs, IP addresses, and registry keys. Moreover, high-level vulnerability descriptions (CVEs), as well as documents describing tactics, techniques and procedures (TTPs), attackers, tools and mitigations for given attacks are widely shared types of CTI. With the increasingly complex nature of threats, these latter high-level textual descriptions of attacks became more prevalent in the form of reports often published as white papers and blog posts.

While these reports are crucial in understanding threat developments, they often must be interpreted by a human to extract this knowledge and transform it into structured data that security systems can understand. Scholars and industry have proposed various models, standards, and ontologies to represent the concepts and semantic relations of these high-level descriptions as knowledge graphs [113, 146–148, 155, 167, 199]. The STIX 2.1 [155] format has widely been adopted as a standard format by many organizations to describe cyber threat intelligence as it provides a generic and extendable framework. However, many other, more specific models and ontologies exist (often expressed in STIX 2.1), such as used by MISP [167] to express objects clustered into galaxies and MITRE's ATT&CK [199], D3FEND [113], CAPEC [146], CWE [148] and CVE [147] frameworks. Organizations use such knowledge graphs for example to link detected IoCs to specific types of threats to find appropriate mitigation strategies.

The problem is that most shared CTI documents are not annotated with structured information. Therefore, while these documents are a flexible form for humans to share and understand security threats, they have to be manually analyzed to extract knowledge that can be used by automated security systems. As this is currently a laborious manual process, we would ideally require a system that automatically interprets CTI documents and extracts knowledge in a structured form that can be used by machines to reason about the provided CTI.

Nowadays, natural language processing (NLP) has shown incredible capabilities in knowledge extraction [72, 170]. Hence, many security researchers have proposed solutions to analyze security texts using these NLP techniques [88, 89, 107, 128, 130, 184, 185, 225]. However, due to the rapid advances in NLP techniques and their use in the domain of security, it becomes difficult to understand the difference between approaches, identify why these approaches work, and reason which further steps must be taken to improve the knowledge extraction from security texts. To solve these challenges, our work makes the following contributions:

- **Systematization of Knowledge** by surveying and classifying existing work that uses NLP to perform knowledge extraction from CTI reports, and by providing guidelines and insights into future directions to improve and integrate NLP solutions into security systems.
- **Methodological unified framework** EAGLE to combine and adapt NLP approaches for security-specific texts.
- **Technical implementation** of EAGLE in a customizable plug-and-play framework to bootstrap future research to solve security-specific NLP challenges.

## 2.2 Dissecting NLP

To understand the challenges faced when processing threat intelligence, we dissect the widely accepted generic NLP challenges [136] and common generic solutions through a running domain-specific example. However, first, we define our goal of *threat intelligence processing* as identifying the steps and relations between steps of a cyber attack that are described in security reports such as CTI reports and CVE descriptions. An NLP approach that solves this goal takes as input a security report and extracts knowledge that describes the individual steps of the attack described by the security report. The main objective of such an approach is to identify the:

1. *entities* (e.g., attackers, targets, malicious software),
2. *actions* that each of these entities perform, and
3. *relations* between entities and actions.

As we will show in Section 2.3, approaches may include and adapt standard NLP techniques to identify *entities*, *actions* and *relations* or use data-driven models to classify or generate texts. In any case, understanding the generic challenges in NLP helps to systematize existing work and understand and possibly improve the steps that these approaches take. The NLP challenges discussed in this section form the basis of our plug-and-play framework EAGLE as shown in Figure 2.1.

### 2.2.1 Naive matching

As specified previously, the goal of threat intelligence processing is to identify *entities*, *actions* and *relations* within the text of security reports. A naive way of doing so is by searching for known patterns representing entities, actions and relations between them. However, first, this approach would require a vast database of patterns to deal with all the quirks of natural language. Second, any pattern that is not explicitly defined would be missed. To illustrate this, consider the following text

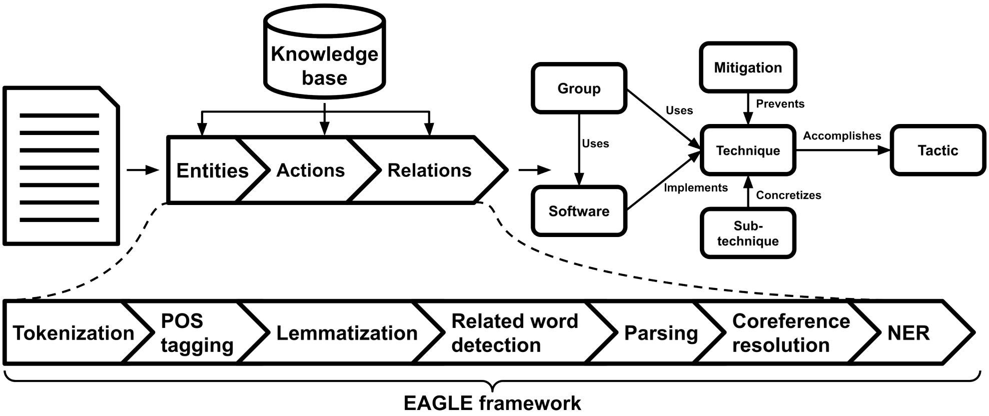


Figure 2.1: **Threat Intelligence Processing.** The proposed EAGLE framework takes security reports as input and uses NLP components combined with a knowledge base to extract entities, actions and relations. In our evaluation, we classify this extracted knowledge into the MITRE ATT&CK framework.

snippet from a security report that we will use as a running example to motivate each step of the NLP framework:

> In one of the Ryuk attacks reconnaissance of the network was done using the tool AdFind by executing a script called "adf.bat". This script was also used in the attack where Hive ransomware was deployed as well as in the Conti attack [...] (Northwave Security)

Next, we search for known entities and action patterns, e.g.:

1. Entities: *Ryuk*, *AdFind*, *Hive*, *Conti*, *ransomware*, **.bat.*

2. Actions: *attack*, *reconnaissance*, *execution.*

Automatically matching these patterns with the security report would lead to several complications due to mismatches in syntax and semantics. E.g., *execution* does not match *execute* (syntax) and we don't want to match *execution* in the sentence "The *execution* of the prisoner."(semantics). Moreover, we have not yet defined relations between entities and actions as it is notoriously difficult to provide an extensive list. We further discuss this and other complications that will arise in the next sections.

### 2.2.2 Tokenization

How should *"adf.bat"* be detected? A human would say the filename is *adf.bat.* However, " is a valid filename character, so should this be included during detection?

Or should we not match at all, because looking at full words separated by spaces, *"adf.bat"* ends in " which does not match our pattern?

These issues arise because simple pattern matching does not perform tokenization [215], the act of identifying individual words, numbers, punctuation, and sometimes prefixes and suffixes as individual tokens. In this work, we also refer to tokenization in the act of detecting sentence boundaries.

Tokenization is mostly considered a solved problem in NLP by maintaining dictionaries of known words and regular expressions such as WordNet [143] and detecting the remaining tokens based on rules for splitting on whitespaces and punctuation.

However, specific cases in security reports, notably indicators of compromise (IoCs), lead to problems when applying these generic techniques in the security domain. E.g., URLs and file paths contain (back)slashes which are often considered separate tokens by generic models, and IPs and URLs contain periods which are normally used to indicate sentence boundaries. In the security domain, these IoCs have a meaning as a token itself and should thus be tokenized differently. Therefore, our framework proposes to extend regular tokenization techniques with regular expressions covering well-known IoCs (e.g., IP addresses, hashes, filenames and paths) that define token boundaries.

### 2.2.3 Part-of-speech tagging

Going back to the first sentence of our example, we may find that *attacks* is found as an action. However, in this case, *Ryuk attacks* is a noun, not a verb, and therefore should not be treated as an action, but would more likely be an entity.

To solve this problem, before trying to identify known patterns, we would like to identify verbs, nouns, adjectives, and any other *part-of-speech* [71]. This allows us to identify verbs as actions, and nouns or proper nouns as entities.

As with tokenization, part-of-speech (POS) tagging is a nearly solved problem for most languages, where current improvements focus on reaching near-perfect prediction [135]. State-of-the-art solutions use features such as capitalization, tense detection, pre- and suffixes [205] as well as inter-word dependencies [204] to assign POS tags.

However, security-specific tokens as found in security reports may exhibit incorrect tagging. E.g., in the sentence:

> The "at" command was used to schedule a task.

The token *at*, is likely misclassified as an adposition because generic NLP techniques often detect closed POS classes such as adpositions through an exhaustive list of options. In this case, generic NLP techniques do not have domain-specific knowledge of tools used by attackers and fail to recognize that in this case *at* should be treated as a proper noun, because it refers to a scheduling tool. In our unified framework, we propose to solve this problem by adding information from a knowledge base such as the MITRE ATT&CK framework [199] to existing POS taggers to improve their performance on security reports.

### 2.2.4 Lemmatization and stemming

In our previous example for detecting *attack*, we did not find an exact match, but instead (loosely) matched the *attacks* token. Nevertheless, when searching a text for the action *attack*, one still expects to find inflected forms i.e., *attacks* and *attacking*. Similarly, searching the conjugated form of *execution*, one expects to find the verb *execute* and its inflected forms *executes* or *executing*.

To allow automated detection, we need to find a base form of each token in the text, which allows us to match it. In NLP, there are two techniques that are related to and attempt to solve this problem: lemmatization and stemming.

Lemmatization uses a combination of rules and exceptions to find the lemma, also known as the *dictionary form* of each token [35, 166]. This has the advantage that meaning is preserved, e.g., *execution* in the sentence:

> The ransomware execution succeeded.

will point to a *different* lemma than in the sentence:

> The prisoner awaits execution.

Because in the first sentence, its meaning is *to carry out*, while in the second sentence, the meaning of *execution* is to *put to death*. Preserving this difference is advantageous when searching for entities and action patterns as we prefer to match tokens with the same meaning. However, lemmatization also restricts itself to tokens with the same POS tags. This means that the verb *execute*, and noun *execution* will have different lemmas. Hence, lemmatization alone is insufficient for performing detection.

In contrast, stemming attempts to overcome this problem by attempting to reduce inflected words or sometimes even derived words to their base form without the need to have a common POS tag or even meaning. Oftentimes, this means that stemming is based on rules, e.g., remove the *-ion* suffix of nouns, or *-ing* suffix of gerunds. Hence, for the noun *execution* and verb *executing* both stems point to *execut* (without the 'e', as it does not have to be a valid word). However, as a disadvantage, both meanings of *execution* in the example above will have the same stem *execut* as well.

We propose to combine lemmatization and stemming techniques with information from a knowledge base such as the MITRE ATT&CK framework to ensure security-related terms can be matched based on their root form while maintaining meaning. This means that we automatically identify all related derivatives for concepts within such a knowledge base, and give them the same stem as the concept in the knowledge base. Additionally, we show that IoCs in security reports require a special form of lemmatization because they are often *defanged*. Defanged IoCs are sanitized in such a way that their representation is different from the original IoC, but to a human, it is still readable as an IoC. Examples are IP addresses or URLs where parts of the address have been modified, e.g., `10.0.0.1` becomes `10[.]0.0.1`. This is done so that signature algorithms don't detect security reports as being malicious themselves and to prevent readers of the report from accidentally accessing those IoCs.

### 2.2.5 Related word detection

While lemmatization and stemming already greatly improve the coverage when identifying entities and actions, they are limited to words that are defined as search patterns. E.g., if the first sentence of our example was changed to:

> In one of the Ryuk attacks exploration of the network was done using the tool AdFind by executing a script called "adf.bat".

a security operator would likely say that the meaning of the sentence remains the same. Oftentimes, it is infeasible to create an exhaustive list of search patterns to cover all possible phrasings for detection. Therefore, we argue any framework used for detecting concepts, should also include the option to detect tokens with a similar *meaning*. This work explores the use of both knowledge bases and unsupervised learning methods to automatically find related words in the text to further increase the coverage of detection (Section 2.4.4).

### 2.2.6 Parsing

Until now, our running example has only dealt with the detection of entities and actions. However, as mentioned previously, we would also like to infer *relations* between these entities and actions to model their interaction. That is, we would like to parse the dependencies [67, 70] of tokens within a sentence to identify the subject and object of the sentence [119, 153]. This would allow us to infer the relation between entities and actions.

While generic NLP methods [51, 104, 129, 154] work reasonably well on text from security reports, performance is improved by better tokenization and POS tagging as essential parts of the NLP framework. Figure 2.2 shows how parsing allows us to infer dependencies within sentences and detect objects and subjects. However, we still require domain-specific knowledge to identify which relations are meaningful to security operators. Hence, we show that parsing methods can be

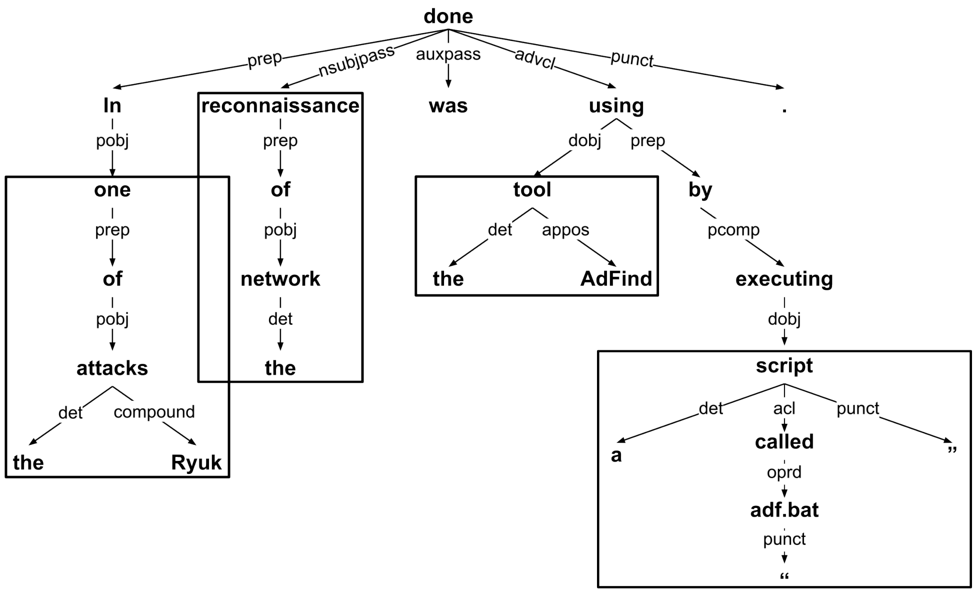


Figure 2.2: **Example of parsed sentence**. When parsing the sentence "*In one of the Ryuk attacks reconnaissance of the network was done using the tool AdFind by executing a script called "adf.bat".*", we find one subject and three objects.

used in combination with knowledge graphs such as the MITRE ATT&CK framework, to determine which relations are meaningful and should be included in the final output.

### 2.2.7 Coreference resolution

The parsing step of the previous example is necessary to infer meanings between tokens within a single sentence. However, security reports often have implicit references between different sentences as well. In the second line of our example, *This script* refers to *adf.bat*. However, by naively analyzing the second sentence by itself, we would miss that both *Hive ransomware* and the *Conti attack* deploy *adf.bat* as well because the script is not explicitly mentioned. Therefore, we require a way to resolve pronouns (e.g., she/he/it) and other referring expressions to make sure that they can be detected as the entities or actions they refer to. In NLP, this task is known as coreference resolution, and it is commonly performed by neural networks [112, 123]. To enhance detection, we propose to apply existing coreference resolution techniques to extend the coverage over entities, actions and relations. This task is rather generic, as long as the POS tags of tokens in the text are correctly attributed. While the task of coreference resolution is vital in increasing the coverage of security reports, we show that a framework does not require a security-specific solution for this problem.

### 2.2.8 Named Entity Recognition

In our running example, we predefined *entities* and *actions* that we would like to identify within security reports. However, the list used in our example is far from complete. Instead, we require a more comprehensive model for locating and identifying *entities*, *actions* and *relations*.

Within NLP, this process of detecting any *entity* (in our case, this includes actions and relations) is called Named Entity Recognition (NER). While there exist generic NER models for detecting entities such as persons, locations or organizations [126], for the most part, and especially in the case of security reports, NER is a domain-specific task. There are many techniques to perform NER, many of which rely on machine learning classification that can be trained on labeled datasets [121, 126]. Unfortunately, for security reports, there currently does not exist any labeled dataset identifying *entities*, *actions* or *relations* that we can use to train a NER classification or detection model.

Therefore, instead, we propose to perform NER by combining the aforementioned framework components with basic search patterns in the form of:

1. regular expressions for IoCs,
2. entities defined in a knowledge base[1], and
3. actions defined in a knowledge base[1].

This allows us to detect predefined *entities* and *actions* accurately in a generic way using proper tokenization, POS tagging, lemmatization and stemming. Furthermore, related word detection and coreference resolution will allow us to extend detection to include tokens with the same meaning. And finally, parsing will allow us to automatically infer the *relations* between entities and actions.

## 2.3 State-of-the-art

In the previous section, we identified the challenges when applying generic NLP solutions to CTI documents. Existing research into NLP for CTI documents already suggests specialized approaches to identify and extract relevant information from security-related texts, overcoming many of these challenges. In this section, we create a taxonomy that classifies solutions into three categories:

[1] In our case the MITRE ATT&CK framework .

Table 2.1: **Framework components implemented by related work.**

| | Type | Approach | Tokenization | POS detection | Lemmatization | Related word | Parsing | Coreference | NER |
|---|---|---|---|---|---|---|---|---|---|
| Semantic | | AttacKG [128] | ❍ | ❍ | ❍ | × | ❍ | ❍ | ● |
| | | Extractor [184] | ● | ● | ● | ❍ | ● | ❍ | ● |
| | | iACE [130] | ❍ | × | × | × | ❍ | × | ● |
| | | ThreatRaptor [89] | ● | ❍ | × | × | ❍ | ❍ | ● |
| | | TTPDrill [107] | ● | ● | × | ● | ❍ | × | ● |
| Hybrid | | CASIE [185] | ❍ | ❍ | ❍ | × | ● | × | ● |
| | | ThreatKG [88] | ● | × | × | × | ● | ❍ | ● |
| | | TIMiner [225] | ❍ | × | ❍ | ● | ❍ | × | ● |
| Data-driven | C | AITI [219] | ❍ | × | × | × | × | × | × |
| | C | Ditdetector [220] | ❍ | × | × | × | × | × | × |
| | C | rcATT [124] | ● | × | ● | × | × | × | × |
| | R | Purba et al. [168] | ● | ● | ❍ | ● | × | × | × |
| | R | SECCMiner [152] | ❍ | ❍ | × | ● | ❍ | × | × |
| | G | BERT-like [18, 72, 134, 172] | ❍ | × | ❍ | × | × | × | × |
| Gen. | | GPT-like [158, 170] | ❍ | × | ❍ | × | × | × | × |

| | | | |
|---|---|---|---|
| × | action is not present or not mentioned. | C | Classification |
| ❍ | action is present, but not domain-specific. | R | Clustering |
| ● | action is present and domain-specific. | G | Generic |

1. **Semantic models** infer the semantic meaning of the CTI texts based on rules and heuristics over token attributes such as POS tags, lemmas, and parsed dependencies.
2. **Data-driven models** use machine learning models to automatically classify tokens, sentences or parts of the text into categories defined by given frameworks or ontologies.
3. **Generative models** are built with the purpose to generate rather than classify text. This class of approaches is relevant for interpreting CTI documents especially when used as a question-and-answering system.

Furthermore, our taxonomy describes to what extent these existing works solve our identified challenges and discuss open problems for each category. Table 2.1 shows the state-of-the-art works classified according to both our taxonomy and the different NLP problems they address.

### 2.3.1 Semantic models

Semantic models focus on named entity recognition by inferring the meaning of words in the text through dictionaries, rule-based heuristics and (automatic) related-word detection. This means that they often start with manually created patterns specifying the entities that should be detected. Semantic models have the advantage that they are explainable and easily fine-tuned due to the specific dictionaries and heuristics that are used for detection. Furthermore, they often have a fine level of granularity in being able to pinpoint the exact phrase or words leading to detection. Finally, semantic models often have a high precision as detected entities are clearly delineated by the model's configuration. The disadvantage of semantic models is that they often struggle with new or loosely defined entities, due to the lack of dictionary entries or rules describing those entities. Furthermore, the fine-grained detection of semantic models is less suited for high-level classification tasks as classes may not be directly visible in the text as individual entities but may be inferred from the overall meaning of the text.

#### Related works

Security-oriented works that employ semantic NLP models focus mainly on detecting IoCs [88, 89, 128, 130, 184, 225], or Tactics, Techniques and Procedures (TTPs) [88, 107, 128, 184, 185, 225] such as those defined by NIST [111] or MITRE [199]. The common objective shared by semantic models is understanding the relations between recognized entities, which is shown by their focus on parsing. This allows them to extract knowledge that details individual attack steps and the relations between them from one or more reports. We distinguish between purely semantic approaches that rely on regular expressions and dictionaries [89, 107, 128, 130, 184] or heuristics [128] and hybrid approaches that perform NER through data-driven classification, but leverage parsing to infer meaning between detected entities [88, 185, 225].

### 2.3.2 Data-driven models

In contrast to semantic models, data-driven models are not imbued with knowledge of domain-specific entities or sentence structures. Instead, they learn which tokens and structures correlate to classes by observing labeled data. To this end, data-driven models learn embeddings (vector representations) for each token in the corpus [141, 164, 210]. Embeddings may be used to identify individual tokens for named entity recognition, or they may be combined to identify and extract entire sentences [72], paragraphs or even documents [124]. Within data-driven models, we distinguish three sub-categories:

1. **Classification models** take token representations and train supervised models to predict texts [124, 219, 220].
2. **Clustering models** perform unsupervised grouping of token representations to find other related tokens or phrases [152, 168].
3. **Generic models** are first trained on generic unsupervised NLP tasks such as prediction of masked token values [72, 134]. The output of these models can be extended for domain-specific downstream tasks [72, 172] such as text classification.

The advantage of data-driven models is that they do not require exhaustive knowledge of tokens and sentence structures used in a specific domain. This means that data-driven models often observe more flexibility when dealing with small changes in expressions that convey the same meaning. Moreover, new tokens and sentence structures can be retrained when new data becomes available. However, the disadvantage of these models is that they often require vast amounts of labeled data to train their increasingly complex models. While training data itself may in some cases be obtained fairly easily, the corresponding labels for accurate classification are often much more costly to acquire [88]. Generic models attempt to alleviate this problem by first training on unsupervised tasks such as language masking or next sentence prediction [72] where the text itself is masked or modified to obtain labels. This way, the language model itself can be trained from the raw texts, not requiring any domain-specific labels, which bootstraps the model to require fewer labeled data points when training its classification model. Nevertheless, for many security-specific tasks, the amount of available labeled data is limited, inhibiting the full potential of data-driven models.

### Classification models

Classification has been used to determine whether a text is relevant to analyze [168, 220], or, once it is determined to be relevant, classified into different TTP categories [88, 124, 185, 219, 225]. Hybrid approaches [88, 185, 225] classify individual tokens or parts of sentences, whereas purely data-driven approaches classify paragraphs or entire documents [124, 219, 220]. While requiring less a priori knowledge of CTI texts, data-driven classification approaches rely on vast labeled datasets, limiting their application.

### Clustering models

Clustering models [152, 168] find relevant tokens or sentences in CTI reports. These can be used to reduce large reports into their most relevant sentences or find tokens/sentences that are related to previously detected concepts. Therefore, these techniques are often used in combination with semantic or data-driven classification models that can detect these relevant sentences in large corpora.

### Generic models

To overcome the requirement of large labeled datasets for data-driven classification models, BERT [72] introduced a generic NLP model trained to predict values of masked tokens in English sentences and predict whether sentence pairs are logical follow-ups. The idea is that, once the neural network is trained on these generic language tasks, one can extend BERT on so-called “downstream tasks” for a domain-specific purpose. AITI [219] builds on top of BERT's output to predict whether sentences include threat intelligence. BERT derivatives such as RoBERTa [134] improve the performance on various downstream tasks. More recently, cyber-specific BERT derivatives such as CyBERT [172] and SecBERT [109] focused on improving BERT's performance on technical cyber threat intelligence language. However, these generic models still need to be fine-tuned on tasks such as identifying relevant entities in threat reports.

### 2.3.3 Generative models

Finally, we have generative models, which take the unsupervised approach of generic data-driven NLP models one step further by predicting possible continuations of text. In the security domain, this can be used to assist security experts in writing reports [173]. However, more recently, generative models have gained attention in identifying relevant information in texts by employing these models in a Question and Answering (Q&A) setting [171]. In this setting, one provides the generative model with a piece of text and a question regarding that text. The generative model then attempts to formulate the most likely continuation, in this case, an answer, to the posed question. Large, generic generative language models such as GPT-3 [46] and ChatGPT [158] have shown interesting results in answering security-related questions about texts.

However, there are severe pitfalls with these models that must be overcome before they can be useful in the security domain. First, due to the black-box approaches that are used to generate new text, it is difficult to assess whether the generated output is meaningful for domain-specific scenarios. Combined with the lack of labels, understanding and adapting generative models to increase performance is almost exclusively limited to providing different training data. Second, existing generative (Q&A) models are built for generic text and therefore it remains to be seen to what extent domain-specific responses are accurate. Therefore, to perform well in the highly specialized context of security, models must be fine-tuned on security-specific inputs. Additionally, existing model architectures are large[2] and require vast amounts of data (beyond what is openly available in the context of security) to train or even fine-tune. Despite being fully unsupervised and therefore not requiring labels, the amount of available domain-specific texts to train these models may be a limiting factor. Finally, text produced by generative models is either written in natural language which has to be interpreted, or the models have to be prompted to output data in structured formats. The former is not a problem when the output should be interpreted by humans, but when the output should be interpreted by machines, it should still be parsed with either a semantic or classification model.

[2] GPT-3 requires training of 175 billion parameters

## 2.4 EAGLE Framework

In Section 2.2 we motivated the different framework components required for the semantic processing of CTI reports. Furthermore, we have seen that state-of-the-art works already take into account many components to increase their performance (Table 2.1). However, most approaches address only a limited set of challenges or use out-of-the-box NLP solutions to implement individual framework components, with the exception of NER. As NER can be strengthened by proper implementation of other components, it is important to ensure every component performs well on domain-specific CTI reports. In this section, we propose a unified framework that attempts to address previous challenges through adjustable components, which in turn allow us to improve existing NER solutions, and even experiment with new ways to perform NER.

### 2.4.1 Tokenization

Tokenization for generic English text uses rules and exceptions for tokenization on how to deal with punctuation, affixes and word boundaries. In security reports, generic tokenization methods work well for most tokens as they are written in natural text. However, IoCs often pose difficulties for generic rules and require special ways of parsing. Depending on the ruleset used for tokenization periods in an IP address or URL may be parsed as the end of a sentence, or a stand-alone token (e.g., `127.0.0.1` is parsed as the seven tokens `[127] [.] [0] [.] [0] [.] [1]`). Similarly, hyphenated IoCs such as `CVE-2021-45105` will be treated as 5 separate tokens `[CVE] [-] [2021] [-] [45105]`.While IoCs could theoretically be processed as multiple tokens, further analysis such as POS detection is simplified when we treat them as a single token. Rather than treating an IP address as a sequence of [`NUM`, `PUNCT`, `NUM`, `PUNCT`, `NUM`, `PUNCT`, `NUM`], intuitively, it makes much more sense to treat it as a single `NOUN`. Hence, to allow for this further processing, we must first identify IoCs as individual tokens.

To detect IoCs as tokens, we create regular expressions for the IoCs[3]. These regular expressions are based on the official specification for the various IoCs. However, these regular expressions alone are not sufficient for security reports. We found, by examining a small sample set of 50 security reports that 13 of them included so-called *defanged* IoCs [48]. By defanging an IoC, writers of security texts try to prevent signature-based security solutions from detecting reports themselves as malicious. E.g., by transforming IoCs such as IP address 127.0.0.1 into 127[.]0.0.1 which will not match any known signature. However, this also means that to detect IoCs, we must deal with defanged IoCs as well. We use the fangs identified by GoodFATR [48]: encapsulating the "`.`", "`:`", and "`@`" by different brackets `[({})]` and/or

[3] Available at `https://doi.org/10.4121/f28346c4-09eb-49e2-bcbc-6bb07bde1970`.

replacing it with the words `dot`/`at` in IPv4, IPv6, URLs, emails and filenames, and replacing the scheme or backslash in URLs (e.g., hxxp:// instead of http://). In Section 2.6.3 we show the increased performance in detecting IoCs.

### 2.4.2 Part-of-speech tagging

Once individual tokens are identified, we want to ensure that each token has a correct POS tag. POS tags are often assigned using a combination of rules for closed class tags and machine learning models that detect open class tokens.

Closed class POS tags can be detected using fixed lists, e.g. adpositions (in, to, during) or rules, e.g., numbers (anything consisting of digits, possibly with a period or comma). Domain-specific tokens that fall into these categories, such as the scheduling software "at" (which is also an adposition), should be listed as exceptions in case the context indicates that they should be treated as a (proper) noun.

Open-class POS tags such as nouns, verbs and adjectives must be inferred from their context and are often detected using machine learning algorithms. While unknown domain-specific tokens that fall into open-class POS tags can generally be detected quite well, there are pitfalls for security-related texts. Specifically, names of software or adversarial groups that are not capitalized (e.g., ftp, ssh, ipconfig or admin@338) may be incorrectly detected as nouns instead of proper nouns. To solve this issue, we automatically create a list of exceptions based on the concepts defined in the MITRE ATT&CK knowledge base and include them in our framework. We do have to be careful here to not be too liberal in assigning different POS tags. For example, the Trojan software Elise is also called Page, however, should we label every instance of *page* as a proper noun, then the page in *web page* will be incorrectly labeled. Hence, we only apply this list of exceptions to known, uncapitalized groups and software. The performance of our improved POS tagging is shown in Section 2.6.3.

### 2.4.3 Lemmatization and Stemming

Now that we have identified the POS tags, we can reduce each token to its base form that we can use for detecting entities and actions within security reports. By reducing each token to such a base form, we increase our coverage during NER. To do so, we extend traditional lemmatization by including all tokens with the same derivational root. This means that rather than reducing a term such as "persistence" to its lemma "persistence", we find all derivationally related words such as "persistent"/"persist" and reduce them all to a common basic form, e.g. "persist". To this end, we search the lexical database WordNet [143] for any token that has a similar derivational root and choose the shortest form of all tokens with the same derivational root as the base.

### Indicators of Compromise

In Section 2.4.1 we paid special attention to tokenizing IoCs and the act of defanging. We introduce two forms of stemming for IoCs. First, a “refanged” version of defanged IoCs (e.g., refanging `127[.]0.0.1` becomes `127.0.0.1`) which can be used to uniquely identify IoCs. Second, a textual description of the IoC type (e.g., the lemma of `127[.]0.0.1` becomes “IP_address”) that is used by related word detection (Section 2.4.4) and named entity recognition (Section 2.4.7) to match tokens.

## 2.4.4 Related word detection

When detecting named entities, it is trivial to detect tokens and phrases that have been explicitly defined. However, we often want to detect tokens and phrases with a similar meaning as well. Using data-driven approaches, tokens with similar meanings can be inferred from large enough labeled datasets. However, as we have seen, these datasets are largely non-existent in the domain of threat intelligence. Therefore, to assist named entity recognition, we propose two methods to identify tokens that are related to explicitly defined tokens. First, a manual dictionary approach, SecNet, lists known synonyms of security-related terms. Second, we automatically infer related words by creating word embeddings for security terms and finding similar embeddings in a trained corpus.

### SecNet

For generic English text, the lexical database WordNet [143] was created that defines semantic relations such as synonyms, hypernyms and hyponyms. While WordNet is a generic database, it does include various security-related terms and relations. However, many terms and especially relations are not relevant in the security context. E.g., WordNet lists the following definitions (in the form of synonym sets) for the term “execution”:

1. Putting a condemned person to death.
2. The act of performing; of doing something successfully; using knowledge as distinguished from merely possessing it.
3. (comp. science) The process of carrying out an instruction by a computer.
4. (law) The completion of a legal instrument (such as a contract or deed) by signing it (and perhaps sealing and delivering it) so that it becomes legally binding and enforceable.
5. A routine court order that attempts to enforce the judgment that has been granted to a plaintiff by authorizing a sheriff to carry it out.
6. The act of accomplishing some aim or executing some order.
7. Unlawful premeditated killing of a human being by a human being.

In this case, we are interested in Definition 3, which includes the term "instruction execution" as a synonym. This is in sharp contrast with e.g., non-security Definition 1 which lists "capital punishment" and "death penalty" as synonyms. We create our own subset of WordNet, called SecNet[4] by automatically selecting all terms that occur both in WordNet and the MITRE ATT&CK framework and manually selecting the synonym sets (i.e., definitions) related to the security context. We identified 3,200 unique terms in the MITRE ATT&CK framework, of which 1,237 contained one or more synonyms in WordNet. For overlapping terms, we found a total of 2,369 matching synonym sets that include a total of 1,428 unique terms in 5,465 synonym relations.

### Word embeddings

Related words defined in SecNet include relations that are true only for *both* generic English text *and* security-specific texts. However, this approach misses relations that are domain-specific but not generally true for English text or relations for words that are so domain-specific that they do not occur in generic databases such as WordNet. An example might be "drop" and "infect" which do not seem to be related at first sight. However, a "dropper" (with the lemma "drop") is a type of Trojan designed to install malware, i.e. to "infect" a machine. Moreover, tokens that are not present in generic databases such as abbreviations for Command and Control: "C&C" and "C2" are related, but are not present in SecNet either. Producing a domain-specific relation database requires costly manual labor to fully cover all possible phrasings of entities that we want to detect in reports in the wild. Therefore, instead, we propose to extend SecNet using unsupervised data-driven related word detection.

Literature uses word embeddings [37, 72, 141, 164] to represent tokens as vectors. The idea is that this vector embeds the meaning of a token and can therefore be used to identify related tokens based on a similarity function, which in language processing is nearly always defined by the cosine similarity:

$$S_{cos}(A, B) = \frac{A \cdot B}{||A|| \, ||B||} \tag{2.1}$$

Word embeddings are initialized as random vectors for each lemma and then learned by training a neural network where the values of these embedding vectors are updated as part of backpropagation. In our work we test three approaches to obtain word embeddings: NLM [37], Word2Vec-CBOW and Word2Vec-Skipgram [141]. Next, we train these word embeddings over our corpus (Section 2.5) and use the cosine similarity (Equation 2.1) to find the top $n$ (in our work $n = 5$) most similar tokens for each token present in the MITRE ATT&CK framework. We consider

[4] Available at `https://doi.org/10.4121/f28346c4-09eb-49e2-bcbc-6bb07bde1970`.

these *n* tokens to be related words which we use in Section 2.4.7 to detect named entities.

### 2.4.5 Parsing

Parsing determines the semantic relations between words within sentences. This allows us to detect objects and subjects within sentences but also dictates whether in the sentence:

"The analyst detected the adversary using network scanning."

it was the analyst who performed the network scanning or the adversary. Save highly ambiguous edge-cases such as the previous example, generic NLP parsing methods trained as classifiers on large corpora [68, 104, 117, 136, 153] seem to perform rather well, even on the technical language used in cyber threat intelligence. This is especially useful because retraining such methods on security-specific data requires dependency-labeled corpora, which to the best of our knowledge do not exist in the domain of cyber threat intelligence. Hence, we propose to use standard NLP parsing techniques, specifically our framework uses the SpaCy parser due to its state-of-the-art performance [104, 153].

### 2.4.6 Coreference resolution

Similar to parsing, most coreference resolution approaches [42, 112, 123, 194] introduce neural networks trained on large datasets of labeled coreferences. These networks rely on word embeddings but may include POS tags and parsed dependencies as additional information to increase their performance. We found that coreference resolution in the domain of CTI is *not* fundamentally different from any other domain as long as entities such as IoCs and domain-specific terms are properly tokenized, POS-tagged, and parsed. Therefore, for coreference resolution, we propose to use standard techniques. In this work, we use SpaCy's coreferee [106].

### 2.4.7 Named Entity Recognition

Using the previously introduced components, we can detect named entities that lie at the core of our knowledge extraction. Note that in this work, we focus on detecting entities and actions described in the MITRE ATT&CK framework. Here, we distinguish two cases: First, single token entities, e.g., "Ryuk" or "reconnaissance" can be detected using lemmas, related words and coreference resolution. Second, subphrases consisting of multiple tokens that do not necessarily occur as fixed subphrases, e.g. "Credential Access" in the phrase "The attacker gained access to the credential database." These latter subphrases often require POS tagging and parsing on top of the techniques used for single token entities.

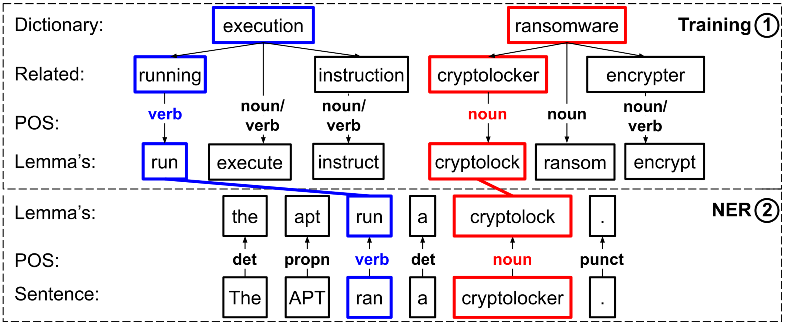


Figure 2.3: **Single token NER.** This example searches for dictionary values "execution" and "ransomware", finds all related words for those values and produces the lemmas for searching. Subsequently, when performing NER on the sentence "The APT ran a cryptolocker.", we compute the lemmas for each token in the sentence and match it against the lemmas in our training dictionary. This detects "run" as a form of "execution" and "cryptolock" as a form of "ransomware".

### Single token entities

Identifying single token entities is relatively straightforward using our previous framework components. Given a dictionary of targets to recognize, we compute all related words for those targets (see Section 2.4.4). Next, we take the lemma and POS of all targets and their related words, as potential matches. Now, when we want to detect these targets in a sample text, we pass the text through our framework, which tokenizes the text and computes all relevant features (POS, lemmas, dependencies, etc.). Finally, we match the (lemma, POS)-tuples in our text to the trained targets to identify named entities. Figure 2.3 illustrates this process of detecting single token entities.

### Subphrase

A more challenging technique is the detection of entities spanning multiple tokens, i.e. a subphrase. Here we have to find a balance between extending single token entity detection to detecting exact (complete and in the same order) subphrases and detecting any occurrence and order of partial subphrases in the text. Detecting exact subphrases is likely to be precise, but would miss entities where the order is changed, or additional words are introduced. E.g., "Privilege Escalation" would not be detected in the sentence "The adversary escalated privileges." (different order) or "The privileges of the adversary were escalated." (contains additional tokens within subphrases). Conversely, being too permissive by allowing any order or sub-phrase to occur separately may lead to a higher recall, but will detect false positives. E.g.,

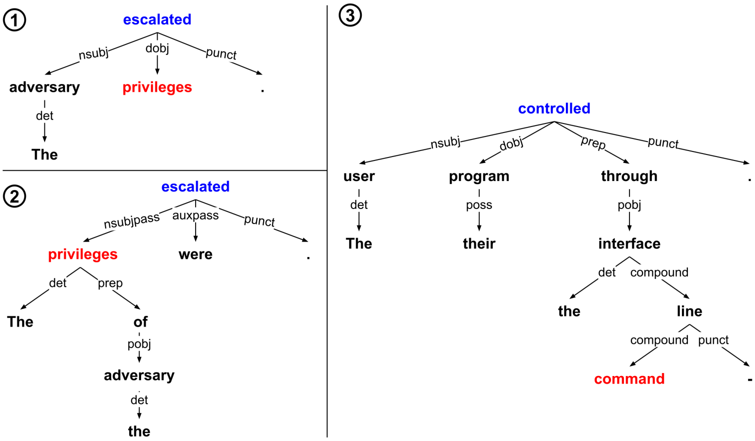


Figure 2.4: **NER parse trees.** Parse trees of the sentences 1) "The adversary escalated privileges."; 2) "The privileges of the adversary were escalated."; and 3) "The user controlled their program through the command-line interface." In cases where we should detect a match (1 and 2), target tokens are *directly* related, whereas non-desired matches (3) do not show a direct relation.

"Command and Control" may be detected in the sentence "The user controlled their program through the command-line interface." Therefore, we propose to leverage information of POS tags and parsed dependencies to more accurately detect subphrases.

Figure 2.4 shows the parse trees of our three examples. Here we find that subphrases that should be detected more often show a *direct* relation in their parse three, whereas false positively detected subphrases often do not show direct relations. We propose to leverage direct dependencies to detect subphrases and discard indirect and non-dependent subphrases for detection. Naturally, one can apply heuristics based on the distance of subphrases tokens within the parse tree to further optimize recognition, but we consider this beyond the scope of this work.

We note that in our third example, we conveniently left out the token "and" when detecting "Command and Control". We experimentally found that tokens such as "and", "in", and "through" in subphrases contain limited information for detection. Such tokens can easily be identified by their POS tags. Therefore, we propose to limit our NER searches in both target subphrases and parsed dependency trees by excluding tokens with POS tags `ADP`, `AUX`, `CCONJ`, `DET`, `PART`, `PRON`, `SCONJ`. Again, exceptions may apply for given subphrases, so certain subphrases may require tuning which POS tags to include in the search.

## 2.5 Datasets

To evaluate and understand the intricacies of the complete framework, we evaluate it using real-world CTI reports. While there exist many publicly available CTI sources, these sources are scattered across the web, making it difficult to find them. Additionally, these reports come in many different flavors (e.g., (white) papers or blog posts with different layouts), making it difficult to extract their contents in a unified way. Furthermore, these reports are often not labeled with any metadata and are therefore difficult to use for classification or regression tasks. These challenges lead to limited data availability for NLP of CTI reports. For this research, we scraped two publicly-available datasets (see description below) that can kickstart research into CTI text processing by collecting the raw text of 6,732 unique reports[5]. Additionally, we worked with the company Accenture to test the EAGLE framework on their proprietary CTI dataset.

### 2.5.1 MITRE CTI dataset.

MITRE's ATT&CK framework provides an open-access knowledge base of adversary tactics and techniques [199]. This knowledge base is based on real-world observations documented in blogs and white papers of security organizations that are referenced within the framework. We collected all references in the MITRE ATT&CK v10.1 framework for a total of 2858 URLs from 755 different sources. Of all URLs, 2,827 were still accessible at the time of collection and were scraped to obtain the threat reports in plain text form. We randomly selected 10 reports consisting of a total of 2,248 sentences and 33,503 tokens. We manually labeled each token (see Section 2.6.1) in these reports according to whether they (partly) describe a MITRE ATT&CK concept. These labeled reports are used in Section 2.6 to evaluate the performance of EAGLE.

### 2.5.2 ChainSmith dataset.

ChainSmith [226], the previous work on detecting IoCs in CTI reports, collected 4,082 references to CTI reports. Unfortunately, their public dataset only includes the references, but did not include the raw text of the reports. Therefore, we rescraped all references of which 3,905 were still accessible. In addition to the sources, the ChainSmith dataset also includes IoCs that were detected by ChainSmith. We note that this is not the ground truth, but rather the predictions of ChainSmith. To allow for a better evaluation, we randomly selected 50 reports from this dataset

[5] Code, data and labels for this research are available at `https://doi.org/10.4121/f28346c4-09eb-49e2-bcbc-6bb07bde1970`. Due to copyright on CTI reports, we publish only our manually labeled reports and include scripts to automatically collect the other datasets.

and manually extracted all IoCs. We use both the manual ground truth and the IoCs detected by ChainSmith in Section 2.6.3 to evaluate and compare our own IoC detection method. Additionally, we include the ChainSmith dataset in the training of embedders for related word detection.

### 2.5.3 ACCENTURE dataset.

In addition to the MITRE CTI dataset, we collaborated with ACCENTURE to evaluate the performance of EAGLE on their internal CTI reports. They provided us with a total of 271 CTI reports where labels were given to the overall report. Furthermore, 5 randomly chosen reports were manually labeled per token to the corresponding MITRE ATT&CK concept by a ACCENTURE expert, just as with the MITRE CTI dataset. This dataset is used to explore the effect of writing styles in reports on the performance of the framework.

## 2.6 Evaluation

We have proposed a framework that includes the necessary components to perform accurate extraction of MITRE ATT&CK concepts in CTI reports. To evaluate this framework, we created a prototype of EAGLE[6], and use this prototype to gain a deeper understanding of its individual components by answering three research questions:

**RQ 1.1** To what extent does each framework component affect the performance of entity detection?

**RQ 1.2** How and to what extent can we influence the performance of individual components?

**RQ 1.3** To what extent is performance affected in practice on different datasets?

We start our evaluation by describing the setup of experiments in Section 2.6.1. Next, Section 2.6.2 performs an Ablation study to answer **RQ 1.1**. Then, we evaluate individual components to answer **RQ 1.2** in Section 2.6.3. Additionally, we address **RQ 1.3** by measuring the performance on different datasets in Section 2.6.4. Finally, we perform a runtime evaluation of the framework in Section 2.6.5.

### 2.6.1 Experimental Setup

We implemented a prototype of the EAGLE framework in Python based on the SpaCy framework[7]. In our evaluation, we also use the official implementation of

[6] Available at `https://doi.org/10.4121/f28346c4-09eb-49e2-bcbc-6bb07bde1970`.

[7] Available at `https://doi.org/10.4121/f28346c4-09eb-49e2-bcbc-6bb07bde1970`.

TRAM[8] and implement TTPDrill by adapting the EAGLE framework to use the TTPDrill ontology. We perform our main evaluation on manually labeled CTI reports from the MITRE CTI dataset. The full MITRE CTI dataset, Accenture dataset and ChainSmith dataset are additionally used to evaluate EAGLE in specific scenarios and to evaluate individual components.

### Metrics

In our experiments, we focus on precision (Eq. 2.2), recall (Eq. 2.3) and F1-score (Eq. 2.4) which are based on the number of true positive (TP), false positive (FP) and false negative (FN) values. TP values are defined as correctly detected entities; FP as incorrectly detected entities; and FN as undetected entities. We deliberately do not include true negative values and related metrics such as accuracy as this would include any token that is correctly not detected. As the majority of tokens in CTI reports do not refer to MITRE ATT&CK concepts, this would give high results that are meaningless.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.2}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.3}$$

$$\text{F1-score} = \frac{2\text{TP}}{\text{TP} + \text{FP} + \text{FN}} \tag{2.4}$$

Additionally, we note that in some of our experiments, complete ground truth was not available, which is the exact problem that our framework attempts to address. When working with limited ground truth, i.e., when only partial labels are available (Sections 2.6.4), we report all values, but focus only on the recall as precision and F1-score cannot be accurately established.

### Dataset labeling

To evaluate EAGLE we manually labeled a total of 15 reports (10 for the MITRE CTI dataset and 5 for the Accenture dataset) on a per-token basis. Labeling is a highly manual task and may be dependent on the knowledge of the expert performing the labeling. Therefore, we let EAGLE, TRAM and TTPDrill [107] detect MITRE ATT&CK concepts and provided them as suggestions to the expert. Next, the expert accepted, discarded, added and modified the suggestions to the best of their knowledge to create a ground-truth dataset. While this approach may bias expert judgment, it simulates a more realistic scenario in which experts use tools such as EAGLE to assist them during labeling.

[8] https://github.com/center-for-threat-informed-defense/tram

Table 2.2: **Performance EAGLE compared with TRAM and TTPDrill.**

| **Approach** | **Precision** | **Recall** | **F1-score** |
|---|---:|---:|---:|
| EAGLE | 86.17% | 86.84% | 86.51% |
| TTPDrill | 27.10% | 55.01% | 36.31% |
| TRAM | 66.67% | 8.96% | 15.79% |
| TRAM conf. ≥ 0.375 | 75.00% | 4.48% | 8.45% |
| TRAM conf. ≥ 0.500 | 100.00% | 2.99% | 5.80% |

**Assisted labeling assessment.** When labeling the 10 MITRE CTI reports, we used the EAGLE framework, TTPDrill and TRAM to suggest labels. Both EAGLE and TTPDrill identified subphrases or even individual tokens as entities of the MITRE ATT&CK framework. For these cases, we define a true positive if the *exact* subphrase was accepted by the expert labeling. Additionally, for TTPDrill, we ignore missed predictions due to differences in MITRE ATT&CK version used for training (TTPDrill used version 6, and our experiments used version 10.1). Conversely, TRAM performs its detection for each sentence rather than tokens or subphrases. Therefore, for TRAM, we define a true positive if the detected MITRE ATT&CK concept exists in the same sentence as the expert label [9].

Table 2.2 shows the performance of all approaches. For a deeper investigation of TTPDrill, we refer to Section 2.6.3. We do find that TRAM achieves a higher precision than the semantic EAGLE framework when setting an appropriate confidence level. However, it has a low recall, meaning it misses several sentences describing potential ATT&CK patterns. As TRAM is a classification-based approach, it has the advantage that it can be trained by supplying user-provided labels of sentences. In our labeling, we used the pre-trained version of TRAM. We discuss in Section 2.7.1 how classification-based solutions such as TRAM may be integrated into a semantic framework to further improve detection.

### 2.6.2 Ablation study

To measure the influence of individual framework components in the detection process, we performed an ablation study. Here, we ran the full EAGLE framework on the manually labeled MITRE CTI dataset. Additionally, we performed a series of experiments where we disabled a single component of the framework in each iteration, except the tokenization and NER components as they are always required

[9] Because TRAM uses a different tokenizer, sentence boundaries may be different, for these cases we define a true positive as being detected in any of the overlapping sentences between TRAM and expert labeling.

Table 2.3: **Ablation study.** Measuring the effect of individual framework components.

| **Approach** | **Precision** | **Recall** | **F1-score** |
|---|---:|---:|---:|
| EAGLE | 86.17% | **86.84%** | **86.51%** |
| Naive matching | **90.50%** | 60.39% | 72.44% |
| No POS | 85.95% | 83.43% | 84.67% |
| No Lemmatization | 88.05% | 69.90% | 77.93% |
| No Related words | 88.28% | 78.83% | 83.29% |
| No Parsing | 85.84% | 82.48% | 84.13% |
| No Coreference | 86.17% | 86.80% | 86.48% |

to perform named entity recognition. Additionally, we performed a *Naive matching* to show the performance when using only tokenization and NER and provide a baseline for evaluating the added value of each component.

Table 2.3 shows the overall result of this experiment, whereas Table 2.4 gives a more complete overview of the full ablation study, grouped per MITRE ATT&CK category. First, we surprisingly find that precision is marginally higher compared to the full framework when lemmatization and related word matching are removed. We attribute this to the lower recall as entities that require certain framework components for detection are slightly harder to detect, and therefore lower the precision.

Second, we focus on recall, where a lower value indicates missed ATT&CK concepts. We find a significant difference of 26.45% points between the full framework and naive matching.

Upon closer investigation, we found that lemmatization and related word detection contributed to a significant increase in recall for single-token entities (e.g., “execution”, “execute”, “executes”). Interestingly, ATT&CK groups and software were hardly affected by any of these framework components as their writing styles rarely differ between their ATT&CK definition and CTI reports.

Conversely, the POS detection and parsing components were responsible for an increase in multi-token, i.e. subphrase detection. Parsing allows for different token ordering when detecting suphrases. In fact, upon investigation, 65.38% of multi-token subphrases appeared in the text exactly as defined by MITRE. Section 2.6.2 gives more details about this experiment. POS detection assists multi-token subphrase detection as it allows us to exclude tokens that convey less information (e.g., adpositions, conjunctions, determiners, see Section 2.4.7) from detection. This allows us to detect subphrases such as “Exfiltration Over C2 Channel” (T1041) using only “Exfiltration”, “C2” and “Channel”, while leaving out less important parts of the subphrase, in this case “Over”.

Table 2.4: **Ablation study per concept.** The full results of the ablation study, including sub-results per MITRE ATT&CK category.

| | **EAGLE** | | |
|---|---|---|---|
| | **Precision** | **Recall** | **F1-score** |
| Average | 86.17% | 86.84% | 86.51% |
| Tactics | 80.94% | 91.97% | 86.10% |
| Techniques | 84.24% | 82.33% | 83.27% |
| Groups | 99.45% | 95.74% | 97.56% |
| Software | 83.16% | 92.59% | 87.62% |
| Mitigations | 50.00% | 100.00% | 66.67% |

| | **Naive matching** | | | **No POS** | | |
|---|---|---|---|---|---|---|
| | **Precision** | **Recall** | **F1-score** | **Precision** | **Recall** | **F1-score** |
| Average | 90.50% | 60.39% | 72.44% | 85.95% | 83.43% | 84.67% |
| Tactics | 74.23% | 52.55% | 61.54% | 96.54% | 88.32% | 92.25% |
| Techniques | 92.61% | 49.00% | 64.09% | 81.33% | 77.32% | 79.28% |
| Groups | 100.00% | 95.74% | 97.83% | 99.32% | 94.81% | 97.01% |
| Software | 94.19% | 90.05% | 92.07% | 84.93% | 92.59% | 88.59% |
| Mitigations | 66.67% | 28.57% | 40.00% | 60.00% | 42.86% | 50.00% |

| | **No Lemmatization** | | | **No Related words** | | |
|---|---|---|---|---|---|---|
| | **Precision** | **Recall** | **F1-score** | **Precision** | **Recall** | **F1-score** |
| Average | 88.05% | 69.90% | 77.93% | 88.28% | 78.83% | 83.29% |
| Tactics | 77.97% | 67.15% | 72.16% | 79.20% | 76.89% | 78.02% |
| Techniques | 86.85% | 61.48% | 71.99% | 87.06% | 73.53% | 79.72% |
| Groups | 100.00% | 95.74% | 97.83% | 99.45% | 95.74% | 97.56% |
| Software | 94.59% | 89.12% | 91.78% | 95.82% | 90.28% | 92.97% |
| Mitigations | 75.00% | 42.86% | 54.55% | 87.50% | 100.00% | 93.33% |

| | **No Parsing** | | | **No Coreference** | | |
|---|---|---|---|---|---|---|
| | **Precision** | **Recall** | **F1-score** | **Precision** | **Recall** | **F1-score** |
| Average | 85.84% | 82.48% | 84.12% | 86.17% | 86.80% | 86.48% |
| Tactics | 80.36% | 86.62% | 83.37% | 80.94% | 91.97% | 86.10% |
| Techniques | 83.44% | 76.10% | 79.60% | 84.23% | 82.26% | 83.23% |
| Groups | 99.45% | 95.74% | 97.56% | 99.45% | 95.74% | 97.56% |
| Software | 92.87% | 93.52% | 93.19% | 92.81% | 92.59% | 92.70% |
| Mitigations | 85.71% | 85.71% | 85.71% | 70.00% | 100.00% | 82.35% |

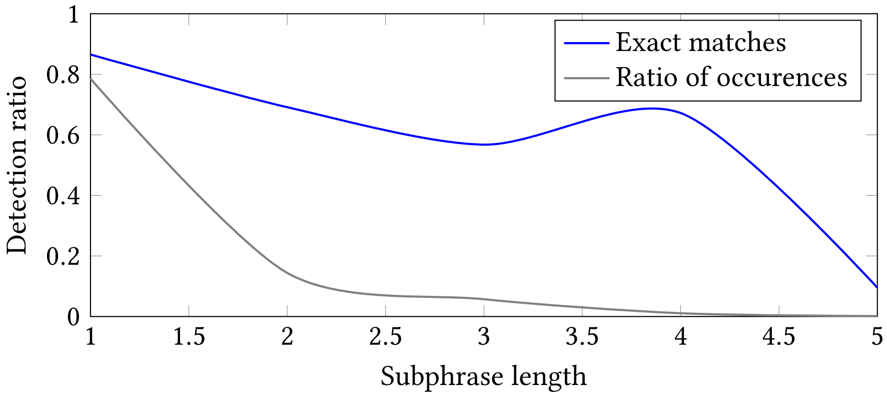


Figure 2.5: **Exact subphrase occurences**. —— gives the % of subphrases that occur exactly as defined by MITRE ATT&CK based on length. —— indicates the % of occurences of subphrases of that length.

### Multi-token subphrases

Figure 2.5 shows the percentage of multi-token subphrases that appeared in the text exactly as defined in the ATT&CK framework as a function of their length. We note that some subphrases rarely differ in ordering (e.g., *Command and Control*), whereas other subphrases can be ordered differently depending on the phrasing of the author (e.g., *process injection* vs. *injects* [into] *process*).

## 2.6.3 Component analysis

The ablation study in Section 2.6.2 suggested that different framework components contribute mainly to increasing recall. In this section, we take a closer look at individual components and how their performance is influenced by other factors.

### Tokenization

The EAGLE framework includes indicators of compromise (IoCs) as individual tokens on top of generic tokenization. To evaluate the performance of this IoC detection, we compare it with ChainSmith on their IoC dataset.

Our IoC detection is based on regular expressions for various IoCs. Unfortunately, to the best of our knowledge, no ground-truth dataset of IoCs occurring in natural text exists apart from the reports that we labeled manually[10]. Hence, to

[10] Existing IoCs, e.g., those shared on the MISP platform do not contain IoCs occurring in natural text. Therefore, this would not provide a realistic scenario.

evaluate the performance of our regular expressions and compare them with ChainSmith, we use the 50 reports from the ChainSmith dataset for which we manually extracted all IoC examples as ground truth. While this evaluation gives us a good indication of the performance of both EAGLE and ChainSmith, the dataset is rather small. Therefore, we perform a second experiment in which we extract IoCs from all 3,905 reports in the ChainSmith dataset using both EAGLE and ChainSmith. Next, we sample 200 IoCs that were detected by both systems, 200 that were only detected by EAGLE and 200 that were only detected by ChainSmith and manually check if these samples are indeed valid IoCs. Finally, we investigate how generalizable this experiment is over larger datasets.

Table 2.5 shows the results of these evaluations. We find that, while ChainSmith has a slightly higher precision on the manual dataset [11], it has a significantly lower recall in both experiments. This means that many of the actual IoCs in the documents go undetected. Upon closer investigation, we found that the main source of missed IoCs by ChainSmith is due to URLs that do not start with a scheme prefix (e.g., `http[s]://`). Our regular expressions still detect URLs as our detection is based on top-level domains instead. Therefore a URL such as `againstvirysscanxp.com` was detected by EAGLE, but not by ChainSmith. Conversely, the incorrectly triggered IoCs mainly consisted of Android application names ending with a valid top-level domain, (e.g., `com.software.app`) which were misclassified as URLs. Due to the similarity of Android application naming conventions and URLs, this is a fundamental limitation of using regular expressions for IoC detection. Nevertheless, EAGLE detects many more IoCs while only having a slightly reduced precision.

**IoC confidence.** Using the binomial proportion confidence interval [213], we extrapolate a confidence interval of the performance for EAGLE and ChainSmith in detection IoCs over the entire dataset. Table 2.6 shows the results of this evaluation. We find that the confidence intervals suggest that the evaluation in Section 2.6.3 generalizes to larger datasets.

### Part-of-speech tagging

While our ablation study showed that POS tagging improves the performance of entity detection, we have not yet evaluated the contribution of adding POS exceptions based on an attached knowledge base. Table 2.7 compares the performance of EAGLE with our ablation study where POS is completely disabled, and EAGLE using SpaCy's generic POS approach. We find that adding knowledge-based POS rules on top of the regular POS component add minimal improvements. This indicates that

[11] Although the 95% confidence interval in Table 2.6 shows that there is a non-negligible probability that EAGLE actually has an equal or higher precision than ChainSmith.

Table 2.5: **Evaluation of IoC detection on ChainSmith dataset.** The Ground Truth experiment used the 50 manually labeled reports as ground truth. During the manual experiment, we extracted all IoCs using EAGLE and ChainSmith and manually classified 600 IoCs (200 detected by both, 200 detected by EAGLE only, and 200 detected by ChainSmith only) to extrapolate the performance. See Table 2.6 for the confidence intervals.

| | **Detector** | **Expected performance** | | |
|---|---|---|---|---|
| | | **Precision** | **Recall** | **F1-score** |
| Ground Truth | Eagle | **95.79%** | **97.52%** | **96.64%** |
| | ChainSmith | 88.07% | 69.98% | 77.99% |
| Manual | Eagle | 96.39% | **99.01%** | **97.68%** |
| | ChainSmith | **98.42%** | 61.41% | 75.63% |

Table 2.6: **95% confidence intervals of Eagle and ChainSmith during manual detection experiment.** Confidence intervals were computed using the binomial proportion confidence interval [213] from the 600 manually classified IoCs.

| **Detector** | **95% confidence interval** | | |
|---|---|---|---|
| | **Precision** | **Recall** | **F1-score** |
| Eagle | 94.10% – 98.68% | 98.83% – 99.18% | 96.41% – 98.93% |
| ChainSmith | 98.15% – 98.68% | 60.13% – 62.69% | 74.58% – 76.67% |

Table 2.7: **Comparison of different approaches to POS tagging, lemmatization and dependency parsing.** We compare the full EAGLE framework, with the framework where components are replaced by a generic NLP component, and with a complete ablation (removal) of each component.

| **Approach** | | **Precision** | **Recall** | **F1-score** |
|---|---|---|---|---|
| EAGLE | | 86.17% | **86.84%** | **86.51%** |
| POS | Generic | 86.13% | 86.80% | 86.46% |
| | Ablation | 85.95% | 83.43% | 84.67% |
| Lemma | Generic | 86.21% | 79.90% | 82.93% |
| | Ablation | **88.05%** | 69.90% | 77.93% |
| Parse | Anywhere | 74.73% | 86.30% | 80.10% |
| | Ablation | 85.84% | 82.48% | 84.13% |

while POS detection is important, it does not seem to be a domain-specific problem and that further improvements should focus on other framework components.

### Lemmatization and stemming

Similar to the POS tagging component, we measured the influence when adding knowledge-based lemmatization on top of SpaCy's default lemmatization by comparing it to the full framework as well as completely disabling the lemmatization component. Table 2.7 shows that the addition of lemmatization rules improves the recall by a notable 6.94% points. This suggests that further adding or refining lemmatization rules could improve the performance of the overall framework.

### Related word detection

As related word detection is a fully custom component, the ablation study already shows its complete influence. Here, we have seen the influence of related words according to a manually annotated subset of Wordnet. However, related words may also be learned using word embeddings. Furthermore, existing approaches such as TTPDrill [107] offer systematic ontologies that can be integrated to find new subphrases describing specific MITRE ATT&CK concepts.

**Embedders.** To find related words, we generate token embeddings for the MITRE CTI and ChainSmith datasets using the Word2vec-CBOW, Word2vec-Skipgram and NLM embedders. Next, we find the closest neighbors of produced embeddings using the cosine similarity to find similar, related tokens. We evaluate the correctness of these found related words by comparing them with related words defined in the relatedwords [175] database. As some tokens in our text do not occur in the relatedwords database due to their domain-specific nature, we limited the evaluation to tokens that occur both in our dataset and the relatedwords database.

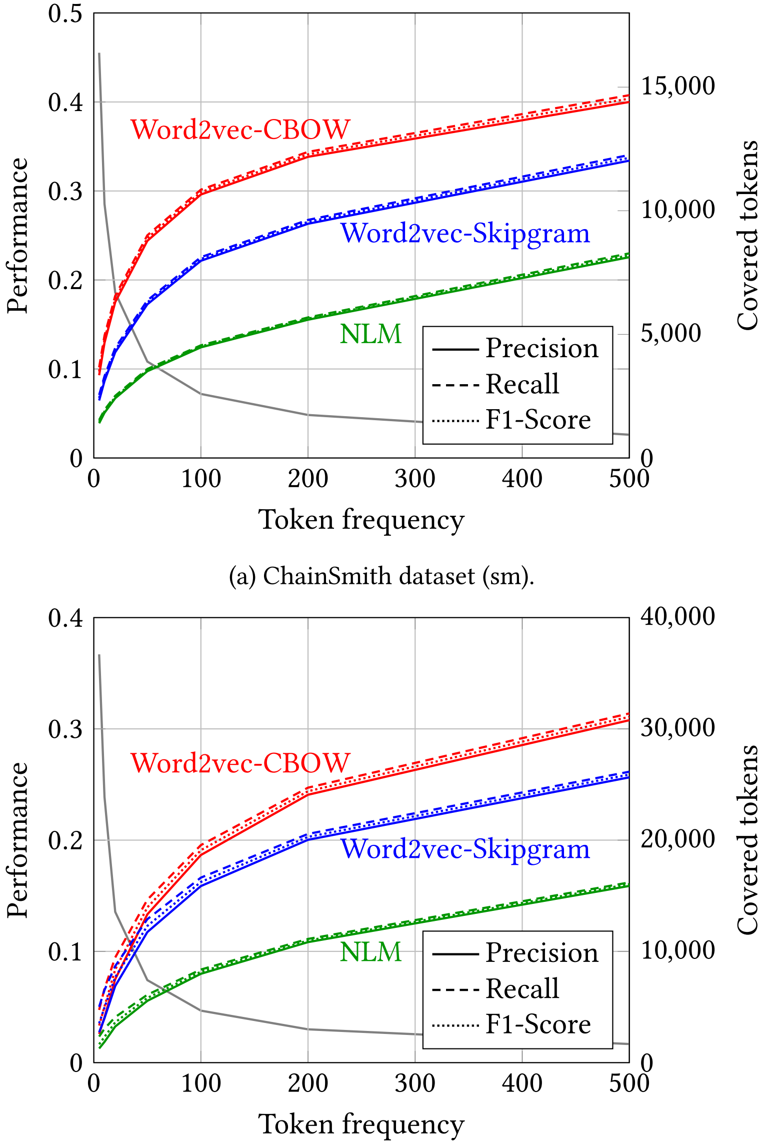
Figure 2.6 shows the result of this experiment when embedders were trained on the (a) ChainSmith and (b) MITRE CTI datasets. While the overall performance is far from good enough to be used in a fully automated way, this Figure does show a trend when tokens occur more frequently. The idea is that the more often a token occurs in the text, an embedder can learn a more accurate representation of this token, and in turn, find related words with better performance. Therefore, future work should investigate the adoption of embedder-based related words when trained on larger datasets. While annotated datasets are sparse in the CTI domain, we recall that embedders do not require labeled reports, but can be trained unsupervised on raw text, lowering the bar for further research.

**Subphrases.** Besides using embedders, state-of-the-art has proposed different ontologies [107] describing TTP, or even automatically mine subphrases that can be used for detection [152]. To measure the effect of using different subphrases for detection, we adapted the EAGLE framework to use subphrases suggested by the TTPDrill ontology instead of the descriptions from the MITRE ATT&CK framework.

Table 2.8 shows the results of using the TTPDrill ontology compared to the descriptions of the MITRE ATT&CK framework. We note that the TTPDrill ontology is limited to 263 ATT&CK techniques from version 6 of the framework, not covering any other ATT&CK categories. Hence, we compare the results to EAGLE using only ATT&CK techniques (equivalent to Techniques in Table 2.9). We find that using different subphrases has a tremendous effect on the performance of the EAGLE framework, both in terms of precision and recall. Therefore, we argue that major care must be taken when creating such subphrases. Interestingly, TTPDrill performed significantly better than EAGLE in detecting T1105 (Ingress Tool Transfer), T1027 (Obfuscated Files or Information), T1192 (Spearphishing Link) and T1193 (Spearphishing Attachment). We believe this is due to the broader coverage of manually defined detection subphrases of TTPDrill that provide it an edge over ATT&CK descriptions. Therefore, we believe that carefully crafted subphrases such as TTPDrill's ontology can improve performance.

### Parsing

In Section 2.6.2, we have shown that parsing offers increased flexibility when detecting multi-token entities compared to detecting exact phrases. However, we do not necessarily have to be so strict to require exact subphrases, instead, we may detect



(a) ChainSmith dataset (sm).

(b) MITRE CTI dataset (sm).

Figure 2.6: **Correct synonyms compared to relatedwords database plotted per token frequency.** When a token occurs more often in the text, embedders have more examples to train with and the found synonyms become better.

Table 2.8: **TTPDrill subphrases.** Compared with TRAM and TTPDrill.

| Approach | Precision | Recall | F1-score |
|---|---|---|---|
| ATT&CK descriptions | 84.24% | 82.33% | 83.27% |
| TTPDrill ontology | 27.10% | 55.01% | 36.31% |

any sentence that includes some combination of tokens in the subphrase. Table 2.7 compares our parsing approach, with detecting subphrases occurring in any form within a sentence, and with exact matching as shown during the ablation study. We find that matching any combination occurring within a sentence almost matches the recall of using our parsed approach, however, this comes at the cost of a significant loss in precision. This is due to incorrectly classified subphrases such as shown in Figure 2.4 (3).

### 2.6.4 Data dependence

We have provided an extensive evaluation using the manually labeled MITRE CTI dataset. However, because this dataset is closely related to the MITRE ATT&CK framework, performance may be skewed. In this section, we explore the performance of the EAGLE framework in more detail per ATT&CK category, and when working with different datasets.

#### Category effects

In Table 2.3 we showed the overall performance of the EAGLE framework. However, performance differed per MITRE ATT&CK category. Table 2.9 details the performance of the EAGLE framework per ATT&CK category. We observe that Groups have high levels of detection due to the limited variance and writing styles. One would expect similar results for Software, however, due to lemmatization, common tokens such as “Internet” are reduced to “net”. While “net” is a software utility mentioned in the ATT&CK framework (S0039), “Internet” should not be detected as such. For both Tactics and Techniques, we observe behavior that can be explained from experiments discussed in Section 2.6.3. Finally, the Mitigations were not mentioned often enough in our reports to give meaningful insights.

#### Accenture dataset

The experiments in Tables 2.3 and 2.9 were performed using the MITRE CTI dataset. Naturally, this data closely matches terminology used in the MITRE ATT&CK framework. To evaluate the effect of different author styles, we asked Accenture to apply

Table 2.9: **Performance EAGLE framework.** Broken down by individual ATT&CK components.

| Approach | Precision | Recall | F1-score | No. samples |
|---|---|---|---|---|
| Average | 86.17% | 86.84% | 86.51% | 2,788 |
| Tactics | 80.94% | 91.97% | 86.10% | 500 |
| Techniques | 84.24% | 82.33% | 83.27% | 1,626 |
| Groups | 99.45% | 95.74% | 97.56% | 189 |
| Software | 83.16% | 92.59% | 87.62% | 463 |
| Mitigations | 50.00% | 100.00% | 66.67% | 10 |

Table 2.10: **Performance of EAGLE over Accenture dataset.**

| Experiment | Precision | Recall | F1-score |
|---|---|---|---|
| Accenture overall | 41.60% | 32.97% | 36.79% |
| Accenture manual | 46.30% | 69.44% | 55.56% |

our framework to their internal CTI reports. We compared the results of our framework to internal labels added by experts for the overall report. Additionally, we asked an expert to sample 5 of these reports and perform a per-token analysis to evaluate how well EAGLE performs for detecting ATT&CK concepts on the most fine-grained level.

Table 2.10 shows the results from this experiment. We see that the performance compared to the MITRE CTI dataset has dropped significantly. We attribute this partly to differences in writing styles, which indicates that NER detection could benefit from additional entity descriptions for improved detection. However, interestingly, manual verification shows that labels produced by our framework for individual tokens achieve a significantly higher recall. This indicates that experts attaching overall labels to CTI reports may have overlooked several ATT&CK concepts, showing the added benefits of automatic threat intelligence processing frameworks in manual labeling tasks. For a further discussion on integrating EAGLE into a security operator workflow, we refer to Section 2.7.

### MITRE ATT&CK descriptions

Another way to evaluate the effectiveness of EAGLE is by feeding it the descriptions of ATT&CK concepts as provided by MITRE themselves. While this is a highly biased experiment, due to overlapping writing styles, it is used in related literature [18] as it allows for easy reproducibility of results.

Table 2.11: **Performance of EAGLE over descriptions of MITRE ATT&CK framework.** The label of the description was used as ground truth. However, descriptions often also contain references to other MITRE ATT&CK concepts. Therefore, in this Table, we focus on recall.

| Experiment | Precision | Recall | F1-score |
|---|---:|---:|---:|
| Average | 21.88% | 93.83% | 35.49% |
| Tactics | 1.58% | 100.00% | 3.11% |
| Techniques | 19.52% | 92.14% | 32.21% |
| Groups | 37.42% | 100.00% | 54.56% |
| Software | 57.56% | 99.47% | 72.92% |
| Mitigations | 10.24% | 29.55% | 15.20% |

Table 2.11 shows the performance of EAGLE on MITRE ATT&CK descriptions. As descriptions often contain references to other tactics, techniques, software or groups which are not included as ground-truth labels, we focus on the recall in this experiment. From the Table, we find that primarily Mitigations and to a smaller extent Techniques were detected with less recall. Upon inspection, we found that this was due to the strictness of our parsing approach. E.g., Mitigation “Operating System Configuration” (M1028) is described as:

> Make configuration changes related to the operating system or a common feature of the operating system that result in system hardening against techniques.

Because the description talks about “configuration changes” rather than “operating system configuration”, EAGLE misses the detection. For a more detailed evaluation and discussion of the parsing component, we refer to Section 2.6.3.

### 2.6.5 Runtime evaluation

We performed a runtime analysis of the 10 labeled reports in the MITRE CTI dataset to investigate the time performance of the framework. For this experiment, we note that each component uses both generic NLP techniques and domain-specific techniques that are added by our framework. Figure 2.7 gives an overview of this runtime analysis of each component within the EAGLE pipeline. We find that the domain-specific additions on top of a generic NLP pipeline increase the time consumption by 4.44%. Moreover, the majority of the time (78.86%) is spent by the

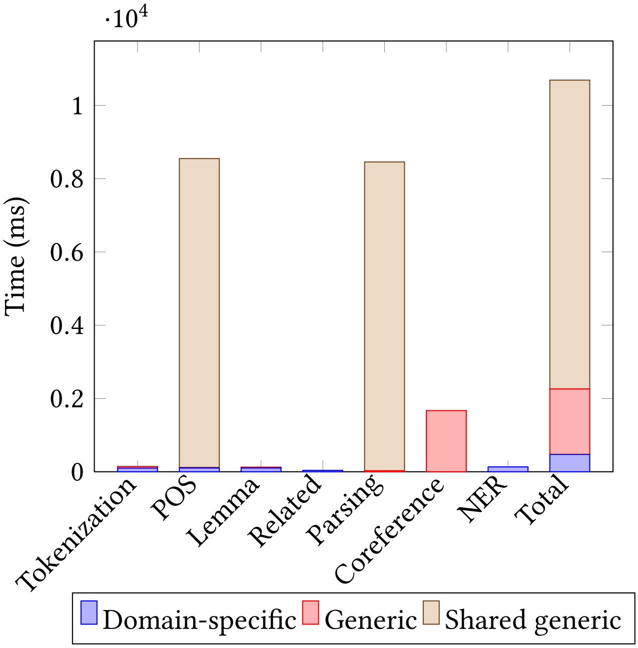


Figure 2.7: **Runtime analysis.** Runtime analysis of different EAGLE components as an average per report over the 10 labeled MITRE CTI dataset reports. We make a distinction between (shared) generic NLP techniques used by the components and domain-specific techniques applied in components.

machine-learning transformer component that assigns POS tags and performs parsing. This transformer is only executed once to create vector representations of tokens that are used by the POS and parsing components to perform their tasks. Faster alternatives to this approach based on heuristics exist[12], but reduce the performance of respective components.

## 2.7 Discussion

### 2.7.1 Semantic frameworks.

We have shown that semantic frameworks such as EAGLE provide an explainable way to identify MITRE ATT&CK concepts in CTI reports. However, large language models (LLMs) such as BERT [72] for classification and GPT [170] for text generation provide interesting opportunities to combine detection performance with the explainability of semantic approaches. Transformer models (the underlying neural

[12] https://spacy.io/models/en

network architectures of BERT and GPT) are already proposed to implement individual framework components such as POS detection, lemmatization and dependency parsing [207]. Moreover, when large amounts of labeled data are available, such transformer models may be trained as classifiers substituting our proposed NER component or as an additional component on top of our NER detector.

### 2.7.2 Data-driven models.

More interestingly, the semantic pipeline components could be used to provide both explainability and potentially improved performance of data-driven approaches. Currently, fully classification-based approaches for performing NER [27] only provide explainability in the form of attention, pointing to the individual tokens in the text that contributed most heavily to a classification decision. Instead, integrating these approaches with a semantic pipeline increases the explainability of these classification models. The current challenge in using data-driven models is that they require large amounts of (labeled) data to be trained with, which is often not available. This is illustrated in our work by the use of embedders for related work detection, which seems promising when trained with a sufficient amount of data. In short, it will be interesting to explore to what extent data-driven classification models can be adapted and integrated with explainable semantic frameworks, especially when they can be trained on large amounts of data. This way, NER performance may be increased, classifiers may use more explainable features, and further processing is still possible due to the availability of other framework components. Hence, combining data-driven approaches with a semantic framework may offer improved performance while still maintaining explainability.

### 2.7.3 Practical application.

In its current form, the EAGLE framework produces helpful suggestions for security experts to identify useful information within CTI reports. However, the performance is not high enough to run fully autonomously. Therefore, we expect the EAGLE framework to be used by experts as an assistant, similar to the way we used EAGLE, TRAM and TTPDrill to create ground-truth labels. The added benefit of running in parallel with a human operator is that missed classifications and incorrect detections may be spotted by humans, and, subsequently be given as feedback to the framework. Operators may introduce exceptions by setting stricter limits on required POS tags, and lemmatization rules or they may add or modify subphrases used for detection. This way, over time, the EAGLE framework can adapt to the specific needs of operators and thereby provide a useful tool for sharing CTI.

### 2.7.4 Extendability.

While the EAGLE framework provides an extensive basis on which to build CTI-processing solutions, it is not complete. Currently, the EAGLE framework identifies MITRE ATT&CK techniques described in CTI reports and uses parsing to find relations between these entities. However, different tasks require different representations and levels of detail within these relations. E.g., generating a knowledge-graph representation of a CTI report may require subject-verb-object (SVO) tuples that connect subject and object nodes through their verb action [182]. Additionally, temporal relations between attack steps are captured using both POS detection and dependency parsing but require additional processing to be incorporated into the framework output. While the EAGLE framework itself does not directly address these potential tasks, it offers the framework components necessary to perform these tasks fostering future research into the processing of CTI reports for various downstream tasks.

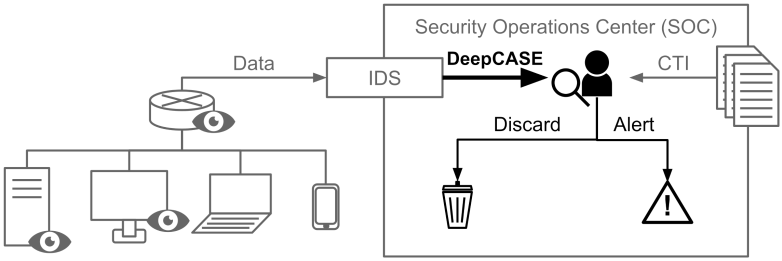
## 2.8 Conclusion

In this chapter, we systematized the use of natural language processing (NLP) for processing Cyber Threat Intelligence (CTI) reports. To understand the components required to perform this task, we provided an extensive background into existing NLP techniques and surveyed state-of-the-art threat intelligence processing papers to identify the NLP challenges they address. Using this knowledge, we proposed a methodological, unified framework EAGLE, that can be used by researchers and industry as a basis to bootstrap future research to solve security-specific NLP challenges. We have demonstrated the effect of different NLP components in the task of knowledge extraction from CTI reports. Moreover, we have provided an in-depth study of the individual components of our framework and showed how they can be modified and improved, to allow others to adapt the framework to specific tasks. By providing our open-source framework at `https://doi.org/10.4121/f28346c4-09eb-49e2-bcbc-6bb07bde1970`, we hope to foster new research and applications in the domain of cyber threat intelligence.

# Chapter 3

## DeepCASE: Semi-Supervised Contextual Analysis of Security Events

Now that we understand the exchange and automatic knowledge extraction from Cyber Threat Intelligence, we look at how this knowledge is used to decide whether detected security events are part of an attack. To this end, we look at automating the initial step in this decision process by answering the following research question:



**RQ 2.** *To what extent can we reduce the manual workload of triaging security events based on contextual information?*

Security monitoring systems detect potentially malicious activities in IT infrastructures, by either looking for known signatures or for anomalous behaviors. Security operators investigate resulting events to determine whether they pose a threat to their organization. Often, a single event may be insufficient to determine whether a certain activity is indeed malicious. Therefore, a security operator frequently needs to correlate multiple events to identify if they pose a real threat. Unfortunately, the vast number of events that need to be correlated often overload security operators, forcing them to ignore some events and, thereby, potentially miss attacks. This Chapter studies how to automatically correlate security events and, thus, automate parts of the security operator workload. We design and evaluate DeepCASE, a system that leverages the context around events to determine which events require further inspection. This approach reduces the number of events that need to be inspected. In addition, the context provides valuable insights into why certain events are classified as malicious. We show that our approach automatically filters 86.72% of the events and reduces the manual workload of security operators by 90.53%, while underestimating the risk of potential threats in less than 0.001% of cases.

This chapter is based on the publication: [1] Thijs van Ede et al. “DeepCASE: Semi-supervised contextual analysis of security events”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 522–539

## 3.1 Introduction

Modern IT infrastructures face constant attacks and are, therefore, continuously monitored. Activities from devices and strategic points within the network are collected and processed by various systems such as Network Security Monitors (NSM, e.g., Zeek [161]) or Intrusion Detection Systems (IDS, e.g., Suricata [85] or Snort [179]). These systems contain *security event detectors* that collect information about security events (e.g., a new contacted host, the use of self-signed certificates, or ports being scanned), which they send to a central Security Operations Center (SOC). In a SOC, events are subsequently triaged by a combination of lower tier security operators and automated rules that combine events into alerts [227]. High priority alerts are then escalated to senior security operators who investigate each alert, and, depending on the threat and impact, take necessary actions [118]. Despite these filtering steps, security operators still have to manually deal with a large number of events and alerts on a daily basis.

The workload of security operators is determined by the quality of the security event detectors. These security systems are expected to flag any suspicious activity to ensure that most malicious activity is detected. However, not all suspicious events are malicious, leading to an overwhelming number of unnecessary alerts for security operators to investigate. To put this into perspective, a 2019 survey by Cisco [58] reported that 41% of 3,540 organizations examined receive over 10,000 alerts per day. Of those alerts, only 50.7% were investigated due to the limited capacity of the security operators, and only 24.1% of investigated alerts were considered an actual attack. A similar 2018 report by Demisto found that companies deal with an average of 174,000 alerts per week, of which only 12,000 were investigated [69]. Moreover, state-of-the-art academic work by Symantec Research Labs uses real-world datasets with an average of 170 security events *per device per day* [191], which shows that even for a few hundred machines, the number of security events easily becomes overwhelming. This illustrates the vast number of alerts that security operators have to deal with. This high workload leads to a condition called *alert fatigue*, where security operators fail to respond to alerts because of the sheer volume they receive each day [99].

In the literature, several works have been proposed to tackle alert fatigue. These works either focus on 1) reducing the number of generated security events by improving individual detectors [56, 101] or 2) prioritizing alerts, a method called *alert triaging* [20, 21, 99]. While reducing the number of security events per detector is useful, it might result in missing a significant portion of the alerts for malicious events. In addition, such a solution needs to be optimized for each detector. As organizations typically use different detectors from various vendors, optimizing at a detector level becomes infeasible.

In contrast, alert triaging shows promising results in reducing the workload of security operators for a more broad range of detectors. Unfortunately, existing triaging approaches still exhibit several limitations. Some works focus on prioritizing individual alerts based on threat indicators associated with each alert [20, 21]. In this case, complex attacks containing many relatively innocent security events remain undetected, while single high impact events are emphasized, despite often being produced by benign processes. More fundamentally, these approaches fail to analyze alerts in combination with other (suspicious) activities in the infrastructure. After all, threat detectors are specialized in finding suspicious behaviors, such as large file uploads or policy violations. However, we argue that suspicious behaviors can, sometimes, be legitimate when viewed together with other activity. To illustrate, consider a scenario where an attacker sends a phishing email containing a link to a website that downloads an executable infecting the machine with a packed botnet malware sample. Security detectors may raise security events for 1) a link to a website with a recently registered domain [212]; 2) a data download using a self-signed certificate [206]; 3) a packed executable [15]; and 4) beaconing activity [105]. When these events occur together, they raise suspicion; however, each individual security event could be benign. After all, these events may be caused by 1) a startup launching its new website; 2) a development webserver where the TLS certificate has not yet been properly initialized; 3) software packed for compression or to protect intellectual property [15]; 4) applications periodically checking servers for updates. It is only when we look at this sequence of events as a whole that we can be confident of an attack. Therefore, techniques prioritizing individual alerts still leave the contextual analysis of alerts up to security operators.

We propose to reduce the workload of security operators by automating the process of analyzing security events from NSM and IDS systems in combination with other triggered events. Figure 3.1 gives an overview of this approach. The core idea is that besides examining the security event itself, we also analyze the security events preceding it. We call these preceding security events the *context* in which an event is triggered. We refer to an event in combination with its context as an event *sequence.* Analyzing the context helps us understand what activity triggered the events. This allows us to better distinguish between an event that is benign (e.g., a self-signed TLS certificate presented by a development server) and a very similar, yet malicious, event in the context of an attack. Moreover, it allows us to present fewer and more detailed alerts to security operators, which drastically reduces their workload.

Using additional information to assess security events is not entirely new. SOC teams frequently use SIEM (security information and event management) products that include hand-crafted expert rules to combine events into alerts for known attack strategies [64]. Other systems, such as Zeek [161], even offer programmable interfaces for security operators to create these rules themselves. However, these

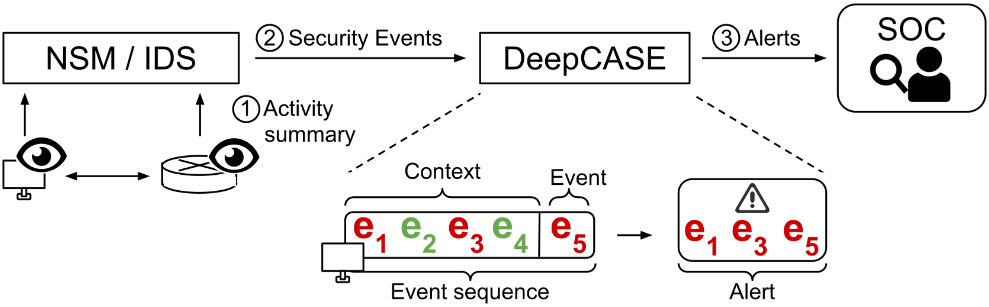


Figure 3.1: **DeepCASE setup.** 1. Agents or probes summarize activity from monitored devices and send it to a network security monitor (NSM) or intrusion detection system (IDS). 2. These NSMs or IDSs contain several detectors that identify security-relevant events (e.g., `Repeated SSH login attempt` or `Observed signature for malware X`). Normally, a security operator would investigate and deal with these events. Our work proposes DeepCASE as an intermediate step. Here, each event is analyzed in a context, defined by the preceding events from the same device. Together they form an event sequence. Our goal is to correlate relevant events within each sequence and present them to the security operator as one alert.

approaches require expert knowledge and cover only event sequences that have been manually defined. Academic work, such as NoDoze [99] and later works [98, 100], attempt to automate this process by leveraging process-level data to reconstruct the activity that triggered an alert. However, this approach relies on process-level information only available in host-based detection systems. In contrast, our work focuses on placing events into context by analyzing only preceding security events. This allows our approach to have wider application, as many detectors (e.g., network-level detectors) or even entire environments (e.g., bring-your-own-device settings) do not have access to process-level information.

Contextual analysis of security events is practical only if it is able to handle 1) *complex relations* within sequences of events, triggered by 2) *evolving* threats, while 3) remaining *explainable* to security operators. Therefore, this chapter specifically addresses these three challenges, which make contextual analysis non-trivial.

**Complex relations.** Malicious activity often involves several steps that can lead to a sequence of events [142]. Conversely, benign applications may accidentally trigger security event detectors leading to false alerts. Since modern devices often run multiple benign applications simultaneously, the actual malicious behavior can easily get lost within this sequence of events. Thus, contextual analysis must be able to identify relevant context events from the complex event sequences received at the SOC.

**Evolving.** Both the attacks and benign applications that trigger security events evolve over time: benign applications get updated, and adversaries develop novel malware and attack techniques. Additionally, companies introduce new detectors or replace old ones to keep up with new threats or gain more insight into the suspicious activity in their IT infrastructure [75]. Ideally, a contextual analysis mechanism should be able to learn new event sequences; and adapt its detection capabilities for new attacks with minimal input from a security operator.

**Explainable.** Finally, reducing the workload of security operators requires filtering alerts and their corresponding events. When filtered incorrectly, we may miss attacks. Therefore, contextual analysis should provide concrete information as to why certain event sequences are discarded whilst others are marked for further investigation.

To reduce the workload of security operators while tackling all three challenges, we introduce DeepCASE. Our approach leverages a deep learning model designed to expose complex relations between both new and previously observed event sequences. These sequences are subsequently grouped based on a similarity function, providing concrete information about why various event sequences are treated the same. Now, the operators need to inspect only a few security event sequences within each group to determine if action should be taken, which significantly reduces their workload. In summary, we make the following contributions:

- We introduce a semi-supervised deep learning approach for explaining and classifying security events in relation to their context.
- We implement this approach in a prototype called DeepCASE, which is a deep learning system for interpreting events within their context.
- We show that DeepCASE is able to reduce the workload of security operators by 90.53% on a real-world dataset containing all security events of 20 international organizations collected over a period of 5 months.

We make both our prototype of DeepCASE and implementations of state-of-the-art work used in our evaluation available at `https://doi.org/10.4121/86c12ba1-7709-45c3-ade3-897552f98ca3`.

## 3.2 Security model

We consider a set of machines and network endpoints that are monitored by a network security monitor (NSM), such as Zeek [161], or an intrusion detection system (IDS), such as Suricata or SnortIDS. First, these systems collect and summarize activities observed within the IT infrastructure (e.g., `HTTP Request from <SRC> to`

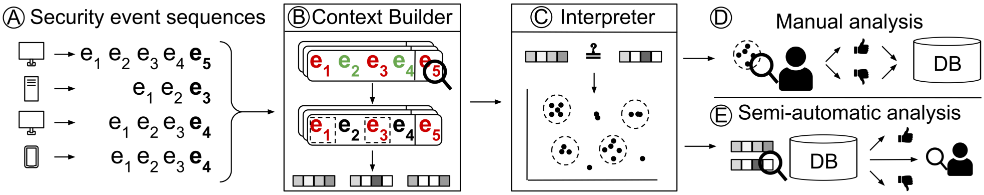


Figure 3.2: **DeepCASE overview of the contextual analysis.** (A) DeepCASE takes sequences of security events from each device as input. (B) Next, the Context Builder identifies relevant contextual security events (represented by the dashed squares), builds an *attention vector* for each sequence, and uses this vector to compute the correlation of each contextual event. (C) The Interpreter compares all correlated contextual events and groups them into similar *clusters*. (D) A security operator analyses and labels the clusters instead of individual events, saving time. (E) Once a security operator labels clusters, similar sequences (based on the combination of contextual events and attention vector) can be automatically classified by comparing them with known clusters.

`<DST> (URI: /index.html)`). When summarizing activities, these systems often already handle application-layer protocols such as SSH, HTTP, and FTP and are able to reconstruct transported files. However, these activities have no notion of being benign, malicious or suspicious, as they simply describe what happens within the infrastructure. On top of these activities, monitoring solutions provide several detectors that detect suspicious *events*, which are sent to a central SOC where a security operator can investigate the event. These detectors may be based on signatures, policies, anomaly detection, or even customizable rules. Examples include `Malicious file download` (signature); `Unusual JA3 fingerprint` (anomaly); or `Self-signed TLS certificate on high port` (policy violation). Figure 3.1 gives an overview of this setup.

Our goal is to analyze all events sent to the SOC and select which events are part of an attack and should be shown to operators. Conversely, events unrelated to an attack should be filtered. After all, a security event may be caused by an adversary attacking the system or may be accidentally triggered by a benign application running on the host. We assume that detectors include information about each host involved in an event, either by leveraging an installed agent (host-based detection), or by deriving the host from the IP address and the logs of a DHCP server (network-based detection), a common assumption in enterprise networks [3].

DeepCASE determines whether an event $e_i$ is part of an attack given the security *context* $[e_0, ...e_{i-1}]$ for this event. The context for event $e_i$ are the $n$ most recent events that occur on the same host as $e_i$, at most $t$ seconds before $e_i$. If there are fewer than $n$ events, the context is simply a shorter sequence.

## 3.3 Approach

We propose DeepCASE to reduce the workload of security operators. Intuitively, our approach searches for correlations within event sequences generated for a specific device. More precisely, we are looking for correlations between events in the context of an event $e_i$ and $e_i$ itself that indicate whether $e_i$ was produced by malicious activity. Once found, we cluster similar event sequences and present them to a security operator who determines whether this combination of events poses a threat to the IT infrastructure. DeepCASE then learns this decision and automatically applies it to similar event sequences found in future event sequences. This semi-automatic approach automatically handles known correlations such that security operators can focus on new threats. Figure 3.2 shows the overview of DeepCASE. First, we take sequences of security events gathered from all detectors, grouped per monitored device in chronological order. Second, for each security event the Context Builder searches for correlations within its context, and captures those relations in what we call an *attention vector*. We note that events from a device may be triggered by different processes or interactions, therefore, a naïve analysis of the context may not find relevant correlations. The Context Builder uses a deep learning model along with an attention mechanism to identify the correlation between events and their context to generate this attention vector. Subsequently, from the attention vector the Context Builder computes the total attention for each contextual event. Third, the Interpreter groups the sequences into *clusters* based on the total attention for each distinct contextual event in a sequence. These clusters can be inspected by a security operator. In manual mode, the security operator classifies each cluster by sampling and inspecting the underlying decision factors (which are provided by the attention mechanism). If the cluster is classified as malicious, the operator can take necessary action for all security sequences within the cluster and, at the same time, the system learns this new cluster. This saves time, as a security operator can assess groups of event sequences rather than individual events. Conversely, when running in semi-automatic mode, DeepCASE compares attention vectors with previously classified clusters and automatically warns the security operator in case a sequence matches a known cluster. By filtering events from the well-known clusters, a large part of the security assessment can be automated.

### 3.3.1 Sequencing events

We first collect all security events generated by detectors in the monitored IT infrastructure. These are then passed to Context Builder, which analyzes chronological sequences of events to identify relevant contextual events and uses them to build *attention vectors*. To reduce the search space for relevant contextual events, we take each event and create a sliding window of $n$ preceding events (in our case 10) from the same device, which form the context. To remove uncorrelated events, we limit the time difference between the event and events in its context to $t = 86,400$ seconds (1 day).

### 3.3.2 The Context Builder

The Context Builder identifies relevant contextual events to build an attention vector. Here, relevance means that our approach should identify events triggered by an attack and discriminate them from events accidentally triggered by benign applications or benign user behavior. To complicate matters, as laid out in our three challenges, the Context Builder should be able to deal with unpredictable, complex relations within event sequences without resorting to a black-box technique. In addition, our approach should be easily updatable to deal with evolving threats and changes in the monitored infrastructure.

To this end, we design a specific kind of recurrent neural network that uses an attention mechanism [33]. Such an attention mechanism is borrowed from the domain of natural language processing (NLP). That domain uses attention to focus a neural network on relevant parts of an input sequence with respect to the desired output [46, 72, 210]. Our work uses this attention mechanism to automatically detect which events in the context $[e_0, ..., e_{i-1}]$ are correlated with the corresponding event $e_i$ in the sequence. Using attention has an advantage over existing state-of-the-art works that use neural networks to analyze sequences of security events [76, 191, 192]: attention can be used to compare the relevance of events in contextual security sequences, which we leverage in our Interpreter[1] (see Section 3.3.3).

Figure 3.3 gives an overview of the Context Builder's network architecture. Normally, this architecture would make a prediction of the expected event $e_i$ by looking at only the context. However, Context Builder is not designed to predict, as we already know the entire sequence of events which occurred on each device. In fact, we train the Context Builder as if it were to predict the event $e_i$ by looking at only the contextual events. If it is indeed able to correctly predict the event $e_i$, we can use the attention vector to analyze which parts of the context were relevant for this prediction. Using such an analysis, the Context Builder identifies contextual events correlated with $e_i$ in the form of an attention vector. When the Context Builder is unable to predict $e_i$, we fall back on the security operator and their existing tools to perform the analysis. This approach uses what is known as an Encoder-Decoder [54] architecture in combination with an attention mechanism [33][2]. At a high level, this means that the encoder network analyzes all contextual events and transforms them into a single vector known as the *context vector*. Next, the attention

[1] For a discussion regarding the use of attention as a means of explaining the relevance of events within a sequence we refer the reader to Section 3.6.

[2] We discuss transformers as an attention mechanism in Section 3.6.

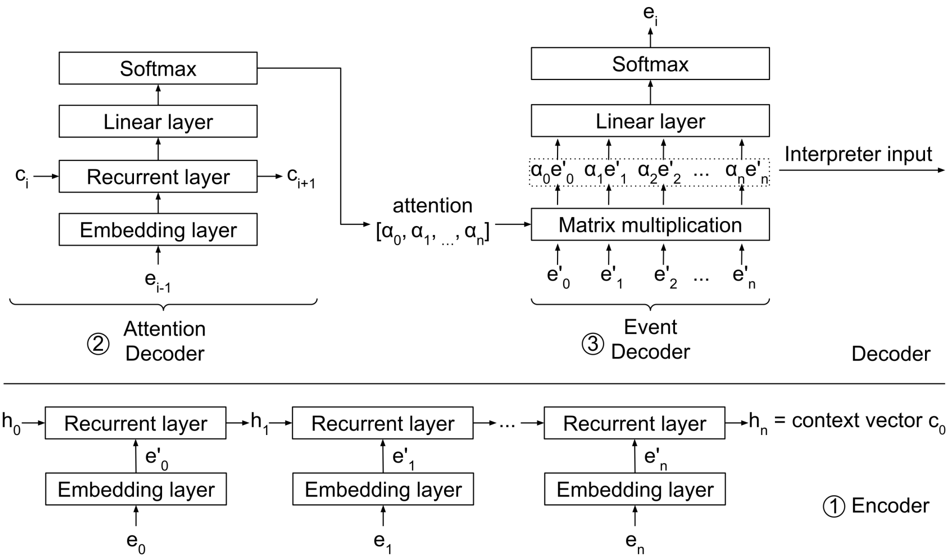


Figure 3.3: **Overview of Context Builder's neural network architecture.** (1) The Context Builder embeds contextual events $S = [e_0, e_1, ..., e_n]$ into embedded vectors $[e'_0, e'_1, ..., e'_n]$. These embeddings are used to generate a context vector $c_0$. (2) The attention decoder then takes this context vector $c_0$, together with optionally generated previous outputs $e_{i-1}$, to learn an attention vector. (3) The event decoder distributes this attention over the embedded inputs $[e'_0, e'_1, ..., e'_n]$, which is used as input for the Interpreter. The modified context that follows from this is then used to predict a probability distribution of the event $e_i$.

decoder takes this context vector and transforms it into an attention vector which specifies a weight (i.e., relevance) of each context event. Then, the event decoder multiplies this weight with the encoded context to obtain the total relevance for all contextual events. From here, the decoder computes a probability distribution over all possible next events. Finally, the system checks whether this "predicted" distribution matches the actual event $e_i$, and passes the total attention for each contextual event to the Interpreter. Section 3.3.3 describes what happens if this prediction is incorrect. Our analysis is described in more detail below.

### Encoder

First, the encoder takes contextual events $[e_0, e_1, ..., e_n]$ as input, which in our case is fixed to $n = 10$ and is left-padded in case there are fewer than $n$ context events. We represent each contextual event as a vector using the embedding layer. Our work embeds the input using one-hot encoding, but embeddings may even be learned by the encoder itself [201]. Next, we use a recurrent layer to combine all inputs into an abstract, internal representation called a *context vector*, which represents the entire context as a single fixed-length vector. Note that the intermediate outputs are discarded, as we require only the final context vector for further computation. The Context Builder uses a single layer of Gated Recurrent Units (GRU) [54] to encode the input into a 128-dimensional context vector. Our empirical results showed that a GRU and Long Short-Term Memory (LSTM) [103] have similar performance, where the GRU is slightly faster.

### Attention Decoder

Now that the input is encoded into the context vector, the Context Builder decodes this vector into an *attention vector* $[\alpha_0, \alpha_k, ..., \alpha_n]$. These $\alpha_k$ values represent the degree to which each corresponding context event $e_k$ contains information regarding the security event $e_i$ in the sequence, normalized to 1. The total attention values for each contextual event are processed by the Interpreter to cluster similar event sequences. In our implementation, the attention decoder takes the context vector as input, passes it through a series of linear layers, and applies a softmax function to normalize the output. Such an architecture allows us to create an attention vector for the event $e_i$.

To describe the context of a particular event $e_i$, we associate each contextual event with its corresponding attention value. To this end, we multiply the attention vector $\alpha$ with the one-hot encoded context sequence $S' = [e'_0, e'_1, ..., e'_n]$, i.e., the matrix multiplication in Figure 3.3. This results in an $n \times m$ matrix of events combined with their attention where $n$ is the size of the context sequence and $m$ is the size of an encoded event. To capture the events in a single vector, we sum all $n$ rows to produce an $m$-dimensional vector describing the context. Recall that the attention values are normalized to 1. Hence, intuitively, this is equivalent to simply summing attention values for each contextual event in case we observe an event multiple times in the context.

### Event Decoder

The attention decoder outputs the attention vector which, once combined with the contextual events, the Interpreter will later interpret and compare with the vectors of other event sequences. However, Context Builder must learn how to decode the context vector into an attention vector. The idea behind this learning process is simple: assuming the attention vector gives relevant context events a higher score than irrelevant events, a neural network must be able to predict the following event $e_i$ given the context events weighted by the attention vector. Therefore, if we train the neural network of Context Builder to predict event $e_i$, we will automatically learn how to assign attention to each contextual event. To this end, the event decoder takes the embedded context events and weights them by performing an element-wise multiplication with the attention vector. Here an attention value of 0 for a context event means that it is ignored in the event decoder, and a value of 1 means that it is the only event that will be considered. Finally, a neural network predicts the probability distribution for the event $e_i$ given these weighted contextual events. This is done with a linear layer of dimension 128 with a Rectified Linear Unit (ReLU) [90] activation function and softmax function to transform the output into a probability distribution.

**Processing event sequences.** Before a neural network can be used for prediction tasks, such as predicting the events from their preceding context events, the network should be trained. Hence, we should show the network an event in combination with its context. In our approach, however, it is important to recall that we are not actually interested in predicting the event, as this is already known to us. Instead, we are interested in the attention vector used to make this prediction possible. This means that we can simply train the Context Builder with the known context as input and the known event as a prediction target. After training the Context Builder with multiple epochs of these inputs, we can use the same contextual events used for training to find their corresponding attention vectors. This makes generation of attention from event sequences an unsupervised process.

In order to train the network, we need to compare its generated output $e_i$ to a desired output $\hat{e}_i$ through a loss function $L(e_i, \hat{e}_i)$. Recall that $e_i$ is a probability distribution over all possible events. Therefore, ideally, DeepCASE should output a probability of 1 for the actual event and 0 for all other events. This can be achieved by using the mean squared error as a loss function, or negative log likelihood when working with log probabilities. However, this incentivizes the network to produce outputs with high probabilities, even when it is not sure that the prediction is correct. To counter this effect, we use label smoothing [200]. Here the desired output $\hat{e}_i$ is a vector where the actual event has a probability value of $1 - \delta$ and the remaining probability $\delta$ is scattered over the other events using their frequency distribution. In this work we use an empirically determined $\delta = 0.1$ (Section 3.5.2). As $e_i$ and $\hat{e}_i$ are now modeled as distributions, we use the Kullback-Leibler divergence [120] as a loss function for the backpropagation.

In short, the Context Builder exploits relations (correlations) between an event and its context to generate an attention vector that assigns a weight to each event in the context. Due to its unsupervised nature, it can easily be updated with new sequences of events [75], making it possible to deal with evolving attack patterns and IT infrastructures. Most importantly, generating the attention values for each contextual event provides the Interpreter with concrete information about which parts of the context are relevant to the corresponding event.

### 3.3.3 The Interpreter

After the Context Builder has computed the attention for each contextual event in the sequence, the Interpreter uses this information to compare different sequences. The idea is that event sequences with similar attention values for the same events can be treated the same way by security operators.

#### Attention query

At this point we should recall that interpreting the attention vector makes sense only if the correct event was predicted. After all, if the Context Builder predicts an incorrect event, interpreting attention would lead to the wrong conclusion. This means that for incorrect predictions, we would fall back on manual inspection, limiting the workload reduction for the security operator. To minimize this effect, we introduce a technique called *attention querying*.

Intuitively, this technique does not ask the Context Builder to predict the security event given its context, but instead asks "given the actual security event that occurred, which attention distribution *would* have resulted in the correct prediction?" We achieve this attention query by temporarily freezing the weights of the event decoder and instead making the attention vector variable. Then we use backpropagation to adjust the attention vector such that the event decoder would result in the highest prediction for the observed event.

**Details.** Traditional attention-based neural networks use attention to focus on specific inputs while predicting an output. Instead, our an attention query[3] asks the neural network "given the actual event that occured, which attention distribution *would* have resulted in the correct prediction?" This is a powerful technique that allows us to automatically learn complex relations within the context of security events.

The attention query uses backpropagation to optimize the attention vector for a given input and output to the event decoder. Mathematically, the event decoder is represented as

$$y = g(f(W\left(\sum_{i=0}^{n} \alpha_i x_i'\right) + b)) \tag{3.1}$$

where $g(\cdot)$ is the softmax function, $f(\cdot)$ is the ReLU activation function, $W(\cdot)+b$ is the linear layer and $\sum_{i=0}^{n} \alpha_i x_i'$ is the matrix multiplication. To find the optimal attention for a given output, we compute the derivative of the loss function $L(\hat{y}, y)$

[3] We provide a pytorch implementation of the attention query at `https://doi.org/10.4121/86c12ba1-7709-45c3-ade3-897552f98ca3`

Table 3.1: **Example sequence of events.** This example shows that the attention will ensure this sequence is clustered in a `Beaconing activity` cluster.

| **Event** | **Attention** |
|---|---:|
| `Login to cryptocurrency mining pool` | 0.0048 |
| `Login to cryptocurrency mining pool` | 0.0048 |
| `Login to cryptocurrency mining pool` | 0.0047 |
| `Login to cryptocurrency mining pool` | 0.0047 |
| `BitCoinMiner` | 0.0043 |
| `BitTorrent` | 0.0050 |
| `Beaconing activity` | 0.2564 |
| `FlyStudio` | 0.0048 |
| `Beaconing activity` | 0.3336 |
| `Beaconing activity` | 0.3769 |

with respect to the variables $\alpha$, representing the attention vector as in Equation 3.2. Finally, we use the Adam optimization algorithm with 100 steps to adjust the attention distribution and find the optimal attention distribution for the observed event. Note that this increases the number of contextual event sequences that we can model. However, there can still be cases where the contextual event sequences do not yield any information regarding the observed event. In these situations, the output probability for observed event $e_i$ remains low and we pass the event to the security operator for further evaluation. For a full evaluation we refer to Section 3.5.5.

$$\frac{\delta L(\hat{y}, y)}{\delta \alpha} = \frac{\delta L(\hat{y}, y)}{\delta g(\cdot)} \cdot \frac{\delta g(\cdot)}{\delta f(\cdot)} \cdot \frac{\delta f(\cdot)}{\delta \alpha x'} \cdot \frac{\delta \alpha x'}{\delta \alpha} \tag{3.2}$$

### Example

In Section 3.5.4, we show that naive clustering does not generalize as well as DeepCASE which includes the Context Builder. To give a concrete example of this difference in performance, consider the following sequence of events as listed in Table 3.1. Here, the attention is heavily focused on the `Beaconing activity`. Therefore, using DeepCASE, it will be clustered in a `Beaconing activity` cluster, whereas the naive clustering approach would place this in a cluster that covers both `Beaconing activity` and `Cryptocurrency`.

### Confidence

It is still possible that even after the attention query, the Context Builder was not able to correctly predict the observed event. We check whether this is the case by comparing the predicted probability of $e_i$ with a confidence threshold $\tau_{\text{confidence}}$ (see parameter selection in Section 3.5.2). If the attention query achieved a sufficient confidence level, we take newly found attention as described in Section 3.3.2. Conversely, if the attention query was not able to pass the confidence threshold, DeepCASE cannot deal with it and passes the sequence to a security operator for manual inspection.

### Clusters

Now that we have modeled each sequence by combining the attention vector with their corresponding events, we can compare and group sequences with similar vectors into a cluster. To this end, we define a distance function between the vectors of total attention per events. Such a function allows us to search for events that occurred in a similar context. We consider two vectors similar if the distance function $d(x, y)$ is smaller or equal to the threshold $\tau_{\text{similarity}}$. The Interpreter defines its distance function in terms of the $L_1$ distance as given in Equation 3.3. The $L_1$ distance is preferable to Euclidean distance because it captures more subtle differences in high-dimensional data. In this work, the dimensionality grows with the number of different possible events $m$. Hence, the $L_1$ distance gives better results.

$$d(x, y) = \|x - y\|_1 = \sum_{i=1}^{m} |x_i - y_i| \tag{3.3}$$

Using this distance function, the Interpreter clusters event sequences using DBSCAN [79]. Here we define the maximum distance $d(x, y)$ between points to be considered part of the same cluster as $\epsilon = 0.1$ (Section 3.5.2). Furthermore, we define the minimum required size of each cluster as 5 (Section 3.5.2). This means that clusters containing fewer datapoints are passed directly to the security operator. Next, each cluster is either passed to a security operator for either manual analysis (Section 3.3.4) or processed further by DeepCASE using semi-automatic analysis (Section 3.3.5).

### 3.3.4 Manual analysis

At this point, each cluster represents a set of sequences of events that share similar contexts. However, we do not yet know whether all sequences within a cluster should be considered benign or malicious, or whether a cluster contains both benign and malicious event sequences. To solve this problem, we present each cluster of similar event sequences to an operator who decides how it should be treated. Section 3.5.5 provides some examples of event clusters generated by DeepCASE.

### Cluster sampling

In the ideal case, all sequences within a cluster should be so similar that analyzing a single sequence is enough to determine whether all sequences are benign or malicious. However, no system is perfect and even an operator may be uncertain about certain events and their context. Therefore, we propose that security operators sample (without replacement) several event sequences from each cluster and analyze them as if they were normal alerts. Next, the operator classifies each sequence into benign vs malicious, or using different classification systems such as risk levels, e.g., `LOW`, `MEDIUM` or `HIGH`. When all sampled sequences fall into the same category and the sample size is large enough, we can confidently treat all sequences in a cluster the same way. Furthermore this classification can be stored into a database that we can use to semi-automatically classify future event sequences (Section 3.3.5). Conversely, if sampled sequences from the same cluster fall into different categories, we know that this cluster is *ambiguous* and will need to be inspected completely by a human operator. By sampling and analyzing a small number of sequences from large clusters, the workload of security operators is drastically reduced. In Section 3.5.3 we evaluate the workload reduction and performance of this sampling process.

### Outliers

We recall that certain event sequences are passed to a security operator because they cannot be handled by DeepCASE. This happens in two situations.

In the first case, the Context Builder did not pass the $\tau_{\text{confidence}}$ threshold. Here we were unable to identify relevant contextual events and our approach does not provide additional benefit. Instead, analysis should be performed by the security operator, falling back on existing analysis tools. In case of new and unknown threats, it is only encouraged that security operators manually inspect them to ensure that no malicious activity slips under the radar. Moreover, the Context Builder is constantly updated with these new event sequences. This means that if they occur more regularly, the Context Builder eventually becomes more confident in identifying relevant contextual events in these previously unknown sequences. Therefore, over time, the number of unidentifiable event sequences will likely decrease.

In the second case, the Interpreter did not find enough similar sequences and manual inspection is similar to sampling from very small clusters. However, in these cases there are only a few items to sample and the resulting classification cannot be generalized to other clusters. The Interpreter may still store these smaller clusters such that when future similar sequences appear, the cluster can still be built incrementally. Moreover, manual analysis can still be facilitated by showing the operator clusters that are outside the similarity threshold $\tau_{\text{similarity}}$, but still have a close similarity to the scrutinized cluster. This provides security operators with more information regarding classification of somewhat-related clusters.

### 3.3.5 Semi-automatic analysis

Once security operators have classified clusters, the Interpreter can automatically compare the vectors of total attention per event of a new sequence (generated by the Context Builder) to these known clusters. If the new sequence matches a known benign cluster, we can automatically discard it without intervention of a human operator. Conversely, when a new sequence matches a known malicious cluster, DeepCASE informs the security operator to take action[4].

However, as we have seen in the Interpreter (Section 3.3.3) and manual analysis (Section 3.3.4), some sequences do not match any cluster. This can either be because 1) Context Builder was unable to achieve a high-enough confidence level for the analyzed event sequence, or 2) because there are not enough other similar sequences to form a cluster. In this case, the semi-automatic analysis notifies the security operator, who evaluates the sequence as described in Section 3.3.4.

## 3.4 Dataset

For our evaluations we use both a synthetic dataset for the reproduction of our results, as well as a large real-world dataset to evaluate the performance of DeepCASE in practice.

### 3.4.1 Lastline dataset

The real-world Lastline dataset consists of 20 international organizations that use 395 detectors to monitor 388K devices[5]. This resulted in 10.5M *security events* for 291 unique types of security events[6] collected over a 5-month period. Events include policy violations (e.g., use of deprecated samba versions, remote desktop protocols, and the Tor browser), signature hits (e.g., `Mirai`, `Ursnif`, and `Zeus`) as well as heuristics on suspicious and malicious activity (e.g., beaconing activity, SQL injection, Shellshock Exploit Attempts and various CVEs). Of the 10.5M security events, a triaging system selected 2.7M events that were likely to be part of an attack. Of these 2.7M likely malicious events, 45.1K security events were confirmed to be part of an attack by security operators, and labeled as `ATTACKS`. These attacks include

[4] In some cases it may even be possible to fully automate the response for known malicious clusters, we discuss this use of DeepCASE as basis for SOAR systems in Section 3.6.

[5] These include devices in a bring-your-own-device setting which were only monitored for a small part of the 5 months. Therefore, the average number of 10.5M/388K = 27.06 events generated per device is significantly lower than the earlier reported 170 events per device per day.

[6] The full list is available at `https://doi.org/10.4121/86c12ba1-7709-45c3-ade3-897552f98ca3`

Table 3.2: **Details of Lastline dataset.** An overview of the number of security events produced per organization, categorized by risk level.

| | | | **Risk level** | | | | |
|---|---|---|---|---|---|---|---|
| **Org.** | **Machines** | **Events** | INFO | LOW | MEDIUM | HIGH | ATTACK |
| 1 | 4184 | 24422 | 24422 | 0 | 0 | 0 | 0 |
| 2 | 13 | 2879 | 2706 | 173 | 0 | 0 | 0 |
| 3 | 50 | 878 | 515 | 341 | 3 | 2 | 17 |
| 4 | 1376 | 8888 | 6553 | 798 | 1473 | 6 | 58 |
| 5 | 386 | 2599 | 1459 | 639 | 395 | 83 | 23 |
| 6 | 229627 | 7117123 | 4858475 | 2191848 | 19215 | 10761 | 36824 |
| 7 | 881 | 132416 | 123841 | 2358 | 2366 | 3848 | 3 |
| 8 | 2185 | 59595 | 57680 | 1430 | 224 | 29 | 232 |
| 9 | 53381 | 319975 | 304394 | 10615 | 658 | 27 | 4281 |
| 10 | 358 | 3337 | 2742 | 501 | 58 | 0 | 36 |
| 11 | 1973 | 81523 | 77316 | 3174 | 179 | 807 | 47 |
| 12 | 1123 | 14867 | 13344 | 1460 | 12 | 1 | 50 |
| 13 | 4607 | 191545 | 180352 | 6855 | 3642 | 134 | 562 |
| 14 | 14 | 1953 | 1883 | 70 | 0 | 0 | 0 |
| 15 | 188 | 26202 | 25927 | 275 | 0 | 0 | 0 |
| 16 | 23 | 382 | 260 | 74 | 27 | 3 | 18 |
| 17 | 18802 | 2062074 | 1850477 | 145995 | 32273 | 30584 | 2745 |
| 18 | 67749 | 340819 | 200296 | 15844 | 124379 | 111 | 189 |
| 19 | 74 | 2789 | 2302 | 483 | 0 | 0 | 4 |
| 20 | 391 | 6521 | 6140 | 373 | 3 | 5 | 0 |

known malware, such as the `XMRig` crypto miner, or remote access Trojans, such as `NanoCore`. Another 46.4K events were classified as a `HIGH` security risk (e.g., successful web attacks and exploitation of known vulnerabilities such as `CVE-2019-19781`); 184.9K events classified as a `MEDIUM` risk (e.g., attempted binary downloads or less exploited vulnerabilities such as `CVE-2020-0601`) and 2.4M events as `LOW` risk (e.g., the use of BitTorrent or Gaming Clients). The remaining 7.8M events were not related to security risks, but were used to give security operators additional information about device activity, and are therefore labeled as `INFO`.

Table 3.2 provides a more detailed description of the Lastline dataset used in our evaluation. This dataset captures security events of 20 international organizations of different sizes.

### 3.4.2 HDFS dataset

We also evaluate DeepCASE on the HDFS dataset [218] used in the evaluation of the related security log analysis tool DeepLog [76]. This dataset consists of 11.2M system log entries generated by over 200 Amazon EC2 nodes. The dataset was labeled by experts into normal and anomalous events, where 2.9% of events were labeled as anomalous. Unfortunately, this dataset lacks metadata about the risk level of security events and is therefore evaluated in terms of workload reduction, but not in terms of accuracy. Despite containing less information, we use the HDFS dataset to provide a reproducible comparison with state-of-the-art systems.

## 3.5 Evaluation

We implemented DeepCASE in Python and compare its performance in workload reduction and performance metrics (e.g., precision, recall, F1-score) with existing workload reduction techniques. Table 3.3 gives an overview of the average workload reduction achieved by all compared methods. Table 3.4 provides an overview of the detection performance, comparing DeepCASE in Semi-automatic mode with all other approaches. Additionally, we evaluate how well DeepCASE deals with the three challenges proposed in the introduction, discuss its robustness against evasion strategies, and perform a runtime analysis to show that DeepCASE is able to handle real-world events generated by major organizations.

### 3.5.1 Setup

To evaluate DeepCASE in a realistic scenario, we split our dataset in a part used to perform manual mode analysis and a part for the semi-automatic mode. The manual mode always precedes the semi-automatic mode, and, therefore, we use the first month of data (2M events) in the Lastline dataset to evaluate our manual mode and the subsequent months to evaluate the semi-automatic mode. The HDFS dataset was split by the original work into training and test sets, which we use in manual mode and semi-automatic mode, respectively. We run all our experiments using the same parameters, which we obtain during a parameter optimization experiment (Section 3.5.2). Unless otherwise specified, we report the average results of 10 runs for each experiment. We followed the three research guidelines for evaluating machine learning-based security systems as detailed in TESSERACT [163]:

- Our experiments have a *temporal training consistency*, meaning that data for our manual evaluation comes strictly before the data used in semi-automatic mode.

Table 3.3: **Workload reduction.** Average workload reduction of DeepCASE compared with existing workload reduction methods. We highlight the *Overall* column which shows the total workload reduction of security operators.

| | | Workload reduction | | | |
|---|---|---|---|---|---|
| | **Method** | **Alerts**[A] | **Reduction**[B] | **Coverage**[C] | **Overall**[D] |
| Manual | DeepCASE | 16, 420 | 99.13% | 94.46% | 93.64% |
| | Cluster N-gram | 35, 640 | 98.12% | 94.62% | 92.83% |
| | Cluster DeepCASE | 45, 400 | 97.68% | 97.96% | **95.69%** |
| Semi-automatic | DeepCASE | 51, 800 | 99.19% | 91.27% | **90.53%** |
| | fully-automatic part | N/A | 100.00% | 86.72% | 86.72% |
| | manual part | 51, 800 | 83.83% | 34.29% | 28.74% |
| | Alert throttling (15 min) | 3, 532, 849 | 49.77% | 100.00% | 49.77% |
| | Alert throttling (30 min) | 2, 889, 607 | 58.92% | 100.00% | 58.92% |
| | Alert throttling (60 min) | 2, 332, 467 | 66.84% | 100.00% | 66.84% |
| | Alert throttling (1 day) | 855, 798 | 87.83% | 100.00% | 87.83% |
| | Rules AlienVault[E] | 421, 693 | 83.78% | 36.97% | 30.97% |
| | Rules Lastline [F] | 299, 246 | 89.49% | 27.02% | 24.18% |
| | Rules Sigma/Zeek[E] | 126, 147 | 92.87% | 25.14% | 23.35% |
| | Cluster N-gram | N/A | 100.00% | 75.70% | 75.70% |
| | Cluster DeepCASE | N/A | 100.00% | 80.59% | 80.59% |

[A] Number of alerts sent to security operators. For DeepCASE and cluster, this is based on 10 sequences per cluster.

[B] Computed as the fraction between alerts and covered events (see Formulas).

[C] Percentage of events covered by alerts (see Formulas).

[D] Total reduction, alerts + uncovered events compared to total alerts (see Formulas).

[E] Based on the event translations provided at `https://doi.org/10.4121/86c12ba1-7709-45c3-ade3-897552f98ca3`.

[F] These rules were used in creating the ground truth (selected events were always shown to analysts) and may therefore give an overly optimistic performance.

**Formulas**

**B** $1 - \frac{\text{Alerts}}{\text{Covered events}}$ **C** $\frac{\text{Covered events}}{\text{Total events}}$ **D** $1 - \frac{\text{Alerts+Uncovered events}}{\text{Total events}}$

Table 3.4: **Detection performance.** Average detection performance of DeepCASE in compared with existing workload reduction methods. We highlight the *Underest.* column which shows how many of the covered events are assigned a risk level lower than their true risk level, which potentially leads to missed attacks.

| | | Performance over covered events | | | | |
|---|---|---|---|---|---|---|
| | **Method** | **Precision** | **Recall** | **F1-score** | **Accuracy** | **Underest.** |
| | DeepCASE | 96.39% | 91.47% | 93.41% | 91.47% | **< 0.01%** |
| | fully-automatic part | 96.39% | 91.47% | 93.41% | 91.47% | **< 0.01%** |
| | manual part | N/A | N/A | N/A | N/A | N/A |
| Semi-automatic | Alert throttling (15 min) | 98.08% | 98.04% | 98.04% | 98.04% | 0.79% |
| | Alert throttling (30 min) | 97.92% | 97.90% | 97.90% | 97.90% | 0.97% |
| | Alert throttling (60 min) | 97.83% | 97.83% | 97.83% | 97.83% | 1.11% |
| | Alert throttling (1 day) | 97.47% | 97.49% | 97.49% | 97.47% | 1.34% |
| | Rules AlienVault[E] | 99.64% | 99.63% | 99.63% | 99.63% | 0.16% |
| | Rules Lastline [F] | 100.00%[F] | 100.00%[F] | 100.00%[F] | 100.00%[F] | **0.00%**[F] |
| | Rules Sigma/Zeek[E] | 99.55% | 99.51% | 99.52% | 99.51% | 0.17% |
| | Cluster N-gram | 96.11% | 94.00% | 94.59% | 94.00% | 0.01% |
| | Cluster DeepCASE | 95.77% | 91.25% | 92.80% | 91.25% | 0.01% |

[E] Based on the event translations provided at `https://doi.org/10.4121/86c12ba1-7709-45c3-ade3-897552f98ca3`.

[F] These rules were used in creating the ground truth (selected events were always shown to analysts) and may therefore give an overly optimistic performance.

Table 3.5: **Parameters.** Values of all parameters used in DeepCASE.

| **Subsystem** | **Parameter** | **Value** | **Section** |
|---|---|---|---|
| Sequencing events | Sequence length | 10 | 3.5.2 |
| | Sequence time | 1 day | 3.5.2 |
| Context Builder | hidden dimension | 128 | 3.5.2 |
| | $\delta$ | 0.1 | 3.5.2 |
| Interpreter | $\tau_{\text{confidence}}$ | 0.2 | 3.5.2 |
| | $\epsilon$ | 0.1 | 3.5.2 |
| | minimum sequences | 5 | 3.5.2 |

- Data should be collected over a *consistent time window*, i.e., there should be no major gaps between collection of data. Our Lastline dataset was collected over a continuous period of 5 months ensuring time consistency.

- There is a *realistic malware-to-goodware ratio in testing*. This ratio follows from the use of a real-world dataset consisting of events collected from 20 organizations.

During the manual mode, the Context Builder learns to produce attention vectors by training the neural network for 100 epochs and extracting the final attention vector (see Section 3.3.4 for more details). In semi-automatic mode, this training is only performed when updating the Context Builder (Section 3.5.5).

### 3.5.2 Parameter selection

DeepCASE uses the parameters listed in Table 3.5. We determined the values of these parameters by performing a 10-fold grid search on the first 1% of data of the Lastline dataset sorted by time. This first 1% is split 50:50 into training and testing sets, and is used only for the parameter selection, i.e., it is not used in further experiments.

#### Sequencing events

The context is defined by 1) the maximum number of events in the context (length) and 2) the maximum time difference between an event and its context (time). To obtain the optimal values, we performed a 10-fold grid search over length values 1, 2, 5, 10, 20 and time values of 1 minute, 1 hour, 1 day, 1 week. For all combinations, we trained the Context Builder with a hidden dimension of 128 and a $\delta$ of 0.1, and we evaluated whether the threshold for the corresponding event was higher than 0.2

($\tau_{\text{confidence}}$). We found that for all input sizes $\geq 5$ and times $\geq 1$ day, between 95.00% and 95.02% of events were correctly predicted from their given context. Therefore, for both length and time values we chose the middle option of 10 events with a maximum age of 1 day.

### The CONTEXT BUILDER

Similar to sequencing events, we performed a 10-fold search for the hidden dimension of the CONTEXT BUILDER, and evaluated whether the threshold for a correctly predicted event reached at least 0.2. Here, we searched powers of 2, $2^1, 2^2, 2^3, ...2^{10}$ and found an optimal value for $2^7 = 128$. The same search over the $\delta$ values of 0.0 to 1.0 with steps of 0.1 yielded an optimal value of 0.1. For the $\delta$ value, the increased performance was mainly due to correct classification of classes with very few samples. This follows from the idea of the $\delta$ value, which increases performance at the cost of a slightly reduced confidence.

### The INTERPRETER

Finally, we used the values obtained from the parameter selection of event sequences and the CONTEXT BUILDER to select parameters for the INTERPRETER. Here we performed a grid search over the $\tau_{\text{confidence}}$ values and $\epsilon$ values, both ranging from 0.1 to 1.0 with steps of 0.1. During this experiment, we measured the overall performance in terms of the F1-score, yielding a $\tau_{\text{confidence}}$ of 0.2 and $\epsilon$ of 0.1. For the minimum sequences, i.e., the minimum number of sequences to be considered a cluster, we chose 5 to give the security operator enough samples to provide a confident prediction. We elaborate on this choice further in Section 3.5.4.

## 3.5.3 Workload reduction

In this section we compare the workload reduction of our approach with existing techniques used by real-world SIEM systems. These techniques include alert throttling and expert rules, as well as more naive, automated methods such as n-gram analysis and our own clustering approach without use of the CONTEXT BUILDER. For each technique, we measure the workload reduction in terms of the percentage of events covered by the raised alerts (coverage), the number of produced alerts compared to these covered events (reduction), and the overall reduction in inspected events (alerts + events not covered) compared to the total number of events analyzed. Furthermore, we discuss the performance over covered events in terms of precision, recall, F1-score, accuracy and percentage of events for which the algorithm underestimated the risk level. Tables 3.3 and 3.4 show the results of all experiments. The remainder of this section discusses how each result was obtained.

### 3.5.4 DeepCASE- manual mode

When used in practice, DeepCASE starts without any knowledge of event sequences. At this stage, DeepCASE runs in manual analysis mode (Section 3.3.4), where all sequences are processed to produce clusters of event sequences (similar events occurring within a similar context). These clusters are then shown to the security operator, who determines if the sequences in each cluster are benign or malicious. In this setting, the workload is reduced because an operator does not have to investigate each individual event; instead, only a small number of samples from each cluster, and the sequences that could not be handled by DeepCASE. We simulated this *manual mode* scenario using the first month of the Lastline events and the training data of the HDFS dataset.

#### Coverage

Table 3.3 shows that for the Lastline dataset, 94.46% of the 2M training event sequences could be grouped into 1,642 clusters. The remaining 5.54% (110.9K) event sequences could not be turned into clusters, either because DeepCASE was not confident enough (95.70%, 106.1K cases) or because there were fewer than 5 other sequences with a similar context (4.30%, 4.8K cases). These remaining sequences can be manually inspected or filtered through existing triage systems, which are complementary to our approach.

**HDFS dataset.** We performed the same evaluation of workload reduction on the HDFS dataset as shown in Table 3.6. In this overview, the alert throttling is left out because the dataset does not contain any timestamp information for the alerts. Additionally, there is no comparison with rulesets as there are no expert-rules available for this dataset. Here, we found similar results where DeepCASE covers 96.39% of sequences with 393 clusters, leading to an overall reduction of 92.26%.

#### Cluster classification

Each cluster contains security events with a similar context. However, this grouping would be useful for the security operator only if each sequence within a cluster is treated the same way. If a cluster contained both benign and malicious samples, or different risk levels, our approach would have a limited benefit. Thus, we scrutinized all event sequences produced by DeepCASE to evaluate to what extent the sequences in each cluster have the same risk classification. We recall from Section 3.4 that the Lastline dataset is labeled into 5 risk categories: `INFO`, `LOW`, `MEDIUM`, `HIGH` and `ATTACK`. Table 3.7 gives an overview of the classification of the clusters. Each risk level details the number of clusters that contain only contextual sequences of that single risk level as well as some statistics about the number of sequences per

Table 3.6: **Workload reduction - HDFS.** Average workload reduction of DeepCASE compared with naive clustering techniques*.

| | Method | Workload reduction<br>Alerts | Reduction | Coverage | Overall |
|---|---|---|---|---|---|
| Manual | DeepCASE | 393 | 95.71% | 96.39% | 92.26% |
| | N-gram | 1,204 | 86.58% | 94.33% | 81.68% |
| | Cluster DeepCASE | 446 | 95.26% | 99.01% | 94.32% |
| Semi-Automatic | DeepCASE | N/A | 100.00% | 96.43% | 96.43% |
| | N-gram | N/A | 100.00% | 93.83% | 93.83% |
| | Cluster DeepCASE | N/A | 100.00% | 98.82% | 98.82% |

*The HDFS dataset does not contain any timestamps, nor are there any rules available. Therefore, we could not compare with alert throttling or rule based approaches.

cluster. Not all clusters contain security sequences of a single risk level. Therefore, we also have a SUSPICIOUS category which captures the clusters containing multiple risk levels. As we can see from Table 3.7, 1,404 of the 1,642 clusters contain sequences of only a single risk level, corresponding to 67.05% of clustered sequences. For clusters containing two adjacent risk levels (e.g., LOW and MEDIUM or HIGH and ATTACK), we find 1,527 of the 1,642 cluster, corresponding to 98.56% of all clustered sequences. Additionally, the maximum cluster sizes and standard deviation values are large. This is because there are many smaller clusters and only a few large ones, i.e., clusters are skewed toward the lower end.

**Cluster distribution.** Clusters produced by the Interpreter vary in sizes as some attacks or benign patterns are more frequent than others. Figure 3.4 gives an overview of the skewed distribution of clusters. We suggest that security operators sample a fixed number of sequences from each cluster. This means that the main body of workload reduction is due to large clusters. Other cluster risk levels show the same skewed distribution.

**Homogeneity.** To measure the extent to which clustered samples belong to the same class (in our case risk level) we use the conventional homogeneity score [180]. This score measures the decrease in entropy of a sample class when the cluster is known. The homogeneity is 0 if all clusters contain multiple different risk levels and 1 for the ideal case where all clusters contain only samples of the same risk

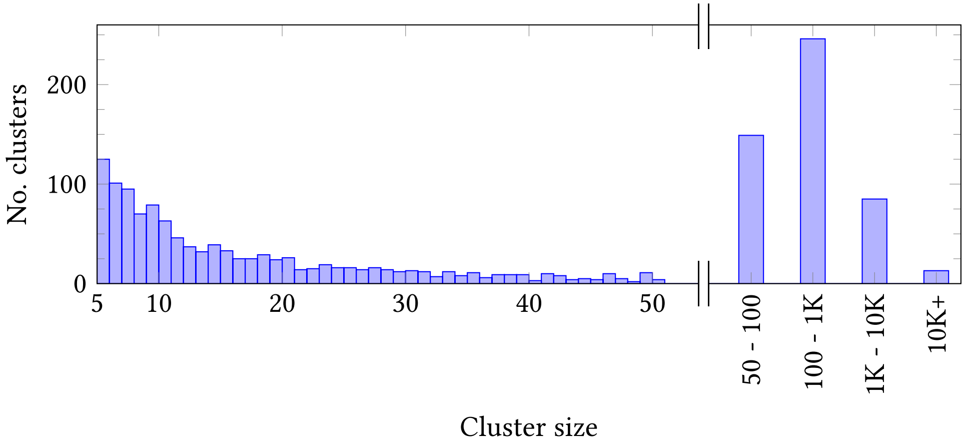


Figure 3.4: **Cluster size histogram of clusters.** Clusters are skewed toward the smaller sizes. This behavior is observed for all risk levels. In general, the resulting workload reduction is most beneficial from the largest clusters.

level. Our clusters show a high quality grouping of sequences of the same risk with a homogeneity score of 0.98.

### Sampling

In the real-world scenario, when a cluster is presented to the security operator, they do not know whether a cluster contains only sequences of a single risk level or multiple risk levels. This is important information as the operator should be able to rely on DeepCASE to group threats with a similar level of risk in the same cluster. During manual mode, the risk level of each cluster must be determined by the operator. To this end, we suggest that the operator samples several event sequences to determine the cluster's risk level. In the ideal case where each cluster has a one-to-one mapping with the risk levels, a security operator would only need to sample a single event sequence per cluster. However, as we have seen in Section 3.5.4, this is not the case. Nevertheless, we can measure the number of investigated samples required for an arbitrary confidence in the risk level of a cluster.

A SUSPICIOUS cluster contains contextual sequences of multiple risk levels. From Table 3.7 we found that 14.5% of clusters are SUSPICIOUS. Given the sequences within a cluster, we can compute the probability of identifying a SUSPICIOUS cluster. We define this as the probability of drawing event sequences of at least two different risk levels, i.e., 1 minus the probability of drawing sequences of only a single risk level. Equation 3.4 gives the probability of detecting a suspicious cluster when sampling $k$ different security sequences. Here $N$ is the total number of event sequences

Table 3.7: **Clusters - Manual mode.** Clusters per risk level. Suspicious clusters contain context sequences with multiple risk levels.

| | | # Sequences | | | | |
|---|---|---|---|---|---|---|
| **Risk level** | **Clusters** | **Total** | **Average** | **Min** | **Max** | **$\sigma$ (SD)** |
| INFO | 1,115 | 1.216M | 1090.3 | 5 | 583.9K | 19.2K |
| LOW | 221 | 41.8K | 189.4 | 5 | 5,557 | 612.9 |
| MEDIUM | 18 | 568 | 31.6 | 5 | 235 | 55.5 |
| HIGH | 17 | 1989 | 117.0 | 6 | 1,107 | 270.6 |
| ATTACK | 33 | 1391 | 42.2 | 5 | 402 | 77.1 |
| SUSPICIOUS | 238 | 619.8K | 2604.4 | 5 | 280.1K | 20.2K |
| Total | 1,642 | 1.881M | 1145.7 | 5 | 583.9K | 17.6K |

within the cluster, $C$ is the set of event sequences from all risk levels and $c$ specifies the set of sequences for each risk level. Note that we model this as sampling without replacement as a security operator will not choose two of the same event sequences to classify. This probability will always be 0 for non-SUSPICIOUS clusters.

$$P(\texttt{suspicious}|k) = 1 - \sum_{c \in C} \frac{\binom{|c|}{k}}{\binom{N}{k}} \tag{3.4}$$

To adopt a conservative approach, a security operator can label a cluster by the highest risk level they have identified from sampling. This way, DeepCASE will miss fewer security threats at the cost of a slightly larger number of event sequences that an operator should investigate. Equation 3.5 gives the probability of sampling at least one event sequence of the highest risk level given a SUSPICIOUS cluster. Here, $|C \setminus h|$ is the number of event sequences from each risk level except the highest risk level $h$.

$$P(\text{highest risk}|k) = 1 - \frac{\binom{|C \setminus h|}{k}}{\binom{N}{k}} \tag{3.5}$$

From Figure 3.5, we find that by sampling 10 samples per cluster, the conservative approach gives a 84.52% confidence for labeling SUSPICIOUS clusters. If a higher confidence is required for the conservative approach, we found that inspecting 95 samples gives a confidence of 95% and inspecting 262 samples gives 99% confidence. However, as only 14.5% of clusters are SUSPICIOUS, inspecting 10 samples per cluster corresponds to a 97.76% overall confidence rating for all clusters.

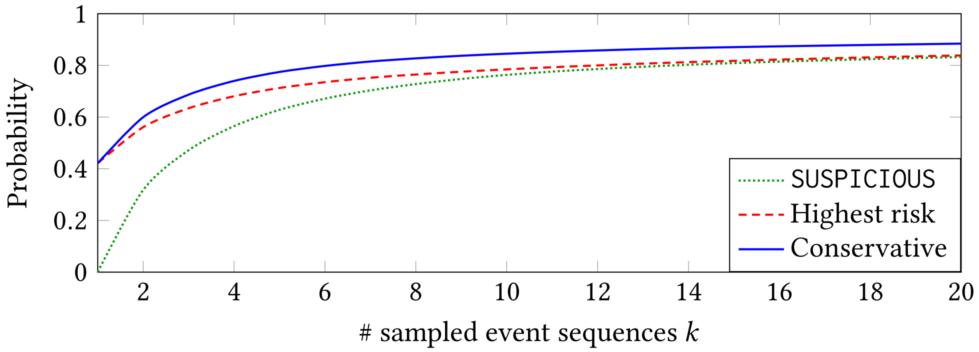


Figure 3.5: **Probability of correctly identifying SUSPICIOUS clusters.** Shows 1) the average probability of identifying SUSPICIOUS clusters (·········); 2) the average probability of finding the highest risk event sequence (----); and 3) the probability of detecting a cluster as SUSPICIOUS, or if not SUSPICIOUS, labeling it as the highest risk sequence (——), i.e. a conservative clustering approach.

### Workload reduction

In short, when running DeepCASE on the Lastline dataset, inspecting 16.4K (10 sequences for 1,642 clusters) event sequences is enough to cover 94.46% (Section 3.5.4) of all security events with a clustering confidence of 97.76%. To cover 100%, i.e., also cover all outliers not handled by DeepCASE, an operator should inspect 127.3K (16.4K clustered + 110.8K outliers) out of 2M events. This reduces the total workload of security operators by 93.64%. For the HDFS dataset, this reduction is slightly smaller, reducing the workload by 92.26%.

## DeepCASE- semi-automatic mode

After a security operator labeled clusters in the manual mode, DeepCASE can be run in semi-automatic mode. During this phase, upcoming sequences are compared against labeled clusters. In case the vectors of total attention per event match, the sequence of events is automatically labeled according to the matching cluster. Depending on the policy of an organization, event sequences labeled higher than a given risk level are escalated to the security operator who can then remove the threat. If the event sequence is found to be benign, it is filtered and not shown to the security operator. Some vectors do not match any of the known clusters, and, as a result, they will either form new clusters or are outliers that will be passed to the security operator for manual inspection. In those cases, the security operator will deal with the sequences as if operating in manual mode, as described in Section 3.3.4.

Table 3.3 shows the performance of DeepCASE on the Lastline dataset running in semi-automatic mode after operating for one month in manual mode. Here, we see that 86.72% of all event sequences match a known cluster, and can be automatically classified. The remaining 13.28% of event sequences is processed in manual mode. From Table 3.3 we see that 34.29% of these non-matching sequences formed new clusters. After this manual step, 8.73% of all sequences could not be clustered, either because they did not pass the $\tau_{\text{confidence}}$ threshold (97.91% of cases) or because there were fewer than 5 samples in a cluster (2.09% of cases). In addition, Table 3.4 shows the performance metrics of DeepCASE. Here we see that the automatic classification of risk levels gives us a reasonable accuracy and F1-score of 91.47% and 93.41%, respectively. However, we must be careful with such numbers, as misclassifying an event sequence with a lower risk means missing attacks. Conversely, misclassifying event sequences as a higher risk is less problematic, and would only give the security operator more manual work. Despite our goal to reduce the workload of security operators, we rather overestimate the risk level at the cost of a smaller workload reduction than miss attacks. The confusion matrix of Table 3.8 shows that the majority of incorrectly labeled cases overestimate the risk level. In fact, DeepCASE underestimates only 47 sequences, which is less than 0.001% of cases.

To understand why DeepCASE underestimates 47 of its semi-automatic predictions, we look at some underestimated cases. Of these underestimates, 3 LOW and 35 MEDIUM risks were classified as INFO, 4 HIGH risks were classified as MEDIUM and 8 ATTACK levels as HIGH risks. The sequences misclassified as INFO are the most undesirable, as these will be ignored altogether by analysts. These sequences were part of 4 different clusters: notably the first detection of the `Bladabindi` backdoor without any prior events was misclassified as `INFO` as well as several unsuccessful `web-application attacks`. However, as detections are often only a single step of an attack, we investigated whether all parts of the attack were misclassified. Here we found that for *all* `ATTACKS` misclassified as `INFO`, at least one earlier or later step of the same attack was classified as `ATTACK`. Other underestimated predictions were less severe and are still shown to analysts, e.g., when a `HIGH` risk was predicted to be `MEDIUM`. In these cases, the incorrect classification is mostly due to similar event sequences observed for different machines. Here the sequences for clusters analyzed in manual mode occurred on machines that were not of vital importance for business continuity. Section 3.6 explores the adjustment of risk level depending on the importance of machines.

### State-of-the-art alert reduction techniques

Instead of raising an alert for each event, many SIEM tools provide options to throttle events as well as options for defining expert rules that aggregate sequences of events into an individual alert. Here, we compare the performance of both methods from state-of-the-art tools with that of DeepCASE.

Table 3.8: **Performance - Semi-automatic mode.** The top row shows the classification performance and coverage of semi-automatic analysis. The bottom shows the confusion matrix of automatically classified samples in the Lastline dataset.

| **Confusion matrix** | | **Predicted** | | | | |
|---|---|---|---|---|---|---|
| | | INFO | LOW | MEDIUM | HIGH | ATTACK |
| **Actual** | INFO | 4896683 | 281528 | 90025 | 132381 | 165 |
| | LOW | 3 | 663327 | 303 | 1 | 1 |
| | MEDIUM | 32 | 0 | 3014 | 14806 | 788 |
| | HIGH | 0 | 0 | 4 | 3419 | 23 |
| | ATTACK | 0 | 0 | 0 | 8 | 12870 |

### Alert throttling

With alert throttling, if an event is triggered multiple times over a given period, only a single alert is shown to an operator, usually in an aggregated form. E.g., suppose the same event X is raised 5 times within the throttling period, the security operator will receive only a single alert after the first event triggered. All subsequent events X within the throttling period are added only as additional information to the first generated alert. After the throttling period passed, a new event will generate a new alert.

We run this throttling mechanism over our Lastline dataset for various throttling periods ranging from 15 minutes to 1 day. Tables 3.3 and 3.4 show the results for this experiment.

The disadvantage of such an approach is that an analyst either has to wait the full throttling period to make a definitive risk assessment of all events within an alert; or she has to assess the risk without having received all events, which potentially misses attacks. The performance metrics for alert throttling show the results after assessing all throttled events with the most common observed risk level.

### Expert rules

Instead of alert throttling, companies often have expert-crafted rules to combine multiple events into a single alert. Sometimes this functionality is embedded into NSM or IDS software. A notable example is Zeek [161], which offers the Zeek Notice Framework, where analysts can write rules that search through Zeek logs and get notified of matches. Such rules can be based on specific sequences or combinations of events related to known common attacks. Depending on the rules, sequences of events that have only a partial match will still be triggered, but often with a lower reliability level.

In this work, we compare the performance of DeepCASE using the Lastline dataset with expert rules from the Lastline; 292 open source rules from AlienVault's OSSIM[7] that cover 82 different known attacks; and the open source rules from Sigma[8] that cover known attacks from various resources such as the MITRE ATT&CK framework. Sigma includes rules that specifically cover Zeek logs and thereby gives a publicly available alternative to Zeek Notices, which are normally specifically written for an organization. As many of the rules from Sigma and AlienVault's OSSIM operate on specific types of detection events, we manually created a bidirectional mapping between all 591 different events used in these rules and the types of events in our Lastline dataset[9]. This allows us to directly apply the Sigma and OSSIM rules to the events in the Lastline dataset and compare their achieved alert reduction to the results from DeepCASE. We have been liberal with this mapping, meaning that any event that could match those provided by AlienVault or Sigma is counted as such. This results in an optimistic coverage of both rulesets and it explains why the coverage of AlienVault is higher than the coverage achieved by the Lastline ruleset. We note that there are many other solutions that provide rule-based detection such as Azure Sentinel [10] and Splunk [11]. For these solutions, we were unable to obtain publicly available rules, and thus could not perform a comparison.

For our evaluation, we count the number of alerts triggered by these expert rules. One single alert consists of all events of a machine (partially) matching one of the available rules. Table 3.3 shows the results of the expert rules. Here, we see that while the reduction for the covered events is similar to DeepCASE, the number of events that are covered is significantly lower. Furthermore, Table 3.4 shows the performance for all events covered by the rulesets is near perfect, with metrics being over 99.51%. This shows that expert rules are highly effective for detecting threats, but still lack much of the coverage that DeepCASE provides. DeepCASE tackles this problem by automatically finding correlations between events, thereby vastly increasing the coverage. Hence, DeepCASE shows much potential to be used in combination with expert-rules for combining events into alerts.

### Naive clustering

DeepCASE's Context Builder detects correlations between an event and its context. To demonstrate why this component is required, we perform an ablation study, i.e., we compare our full approach with a version of DeepCASE without the Con-

[7] https://cybersecurity.att.com/products/ossim

[8] https://github.com/SigmaHQ/sigma

[9] Mappings are available at https://doi.org/10.4121/86c12ba1-7709-45c3-ade3-897552f98ca3

[10] https://azure.microsoft.com/en-us/services/azure-sentinel/

[11] https://www.splunk.com/

text Builder. Additionally, we show the performance increase of our clustering approach compared to clustering on exact matches, i.e., N-grams.

### N-grams

The most straightforward approach for sequence prediction is to treat an event and its context as an N-gram. Here we can store all N-grams in the training data, together with their highest associated risk level and assign the same risk level to N-grams in the test data if there is a match.

Tables 3.3 and 3.4 show the result for this experiment, where each alert is equal to the number of stored N-grams. As with DeepCASE, we assume a security operator will check 10 sequences (N-grams) to determine its risk level. While this approach does not underperform compared to DeepCASE in the manual use case, the semi-automatic use case shows that N-grams do not generalize as well.

### Clustering

Instead of using N-grams, we can approach matching better using clustering as proposed by DeepCASE, without using the Context Builder. This scenario is equivalent to DeepCASE with Context Builder where the attention value for each contextual event is $\frac{1}{n}$.

Table 3.3 shows that the results for using only our clustering approach slightly outperforms DeepCASE in manual mode. However, in semi-automatic mode, it still produces twice the workload compared to DeepCASE with the Context Builder. Therefore, we conclude that the Context Builder generalizes significantly more than naive approaches. This results in DeepCASE roughly halving the workload of a security operator in semi-automatic mode compared to naive clustering.

## 3.5.5 Challenges

This work addressed three challenges that make it difficult to reduce the workload of security operators. Here, we evaluate the extent to which our approach solves these challenges.

### Complex relations

The Context Builder was designed to find correlations between an event and its context. To this end, it analyzes the preceding contextual events and tries to predict what event is most likely to occur next. To understand how well our approach deals with such complex relations, we assess to what extent events are correctly predicted from their context. This prediction based on past security events is not novel, as it was introduced by DeepLog [76] and later extended by Tiresias [191].

Table 3.9: **Prediction results.** Systems trained on first 20% of data and evaluated on remaining 80% of data. Time shows the average amount of time for 1 epoch of training. Best performance is highlighted in **bold**.

| | System | Precision | Recall | F1-score | Accuracy | Train time |
|---|---|---|---|---|---|---|
| HDFS | DeepLog | 89.71% | 89.34% | 89.35% | 89.34% | **1.0 s** |
| | Tiresias | 89.70% | 87.63% | 87.96% | 87.63% | 15.0 s |
| | DeepCASE | **90.41%** | **90.64%** | **90.40%** | **90.64%** | 1.3 s |
| Lastline | DeepLog | 89.65% | 90.40% | 89.82% | 90.40% | **0:06.8 m** |
| | Tiresias | 95.50% | 96.21% | 95.68% | 96.21% | 4:51.5 m |
| | DeepCASE | **97.90%** | **98.06%** | **97.90%** | **98.06%** | 0:13.8 m |

However, both of these works focus purely on the prediction aspect, and cannot be extended to perform the contextual analyses proposed in this work. Therefore, to measure how well DeepCASE is able to deal with complex relations within the data, we compare its prediction performance with the two state-of-the-art systems DeepLog and Tiresias. We implemented both systems as described in their respective papers. While the original source code was not available, upon contacting the authors we received helpful suggestions to re-implement both approaches[12].

We evaluate all three approaches on both datasets described in Section 3.4. We performed 10 training and testing runs for each system, where the first 20% of the datasets were used for training and the remaining 80% was used for testing. Table 3.9 shows the average results of all 10 runs of this evaluation. We see that on both datasets, DeepCASE performs the best in terms of evaluation metrics. The only downside of DeepCASE is that the runtime is slightly slower than DeepLog, but as we show in Section 3.5.7, DeepCASE is easily fast enough for real-world application. In short, DeepCASE shows improvements over state-of-the-art works that were specifically designed to predict future events, while handling their complex relations.

### Evolving event patterns

Our second challenge is dealing with evolving threats and event patterns. DeepCASE should be able to detect new event sequences and group them into new clusters to show to the security operator. In addition, DeepCASE uses a neural network

[12] DeepLog code is available at `https://doi.org/10.4121/7a6086ad-1cd1-4a76-be8a-b7c0b6d17311`.

Tiresias code is available at `https://doi.org/10.4121/4e12761f-716a-4ea6-b08c-a6a6e459893d`.

in the Context Builder to model sequences. We generally expect the results to improve as we show the network an increasingly large number of event sequences. We recall that these sequences are automatically generated, and therefore do not need to be labeled. When monitoring a new IT infrastructure or handling events of new detectors, DeepCASE should quickly be able to update using these new incoming events.

**Online updating.** We demonstrate DeepCASE's ability to evolve with new events. Operating in manual mode, our approach already is able to deal with new events and different event patterns. Therefore, we evaluate the increase in performance of DeepCASE running in semi-automatic mode if periodically updated. In this experiment, we compare the performance between no updates and daily, weekly and monthly updates. In each update, we show new data to the Context Builder and add newly produced and manually labeled clusters to the database of the Interpreter to be used for comparing future event sequences.

Table 3.10 shows the results for this experiment. We find that regularly updating DeepCASE improves its coverage between 5.91 and 6.85 percentage points. The performance in terms of accurately predicting the risk level of these newly covered items is only marginally lower than the original detection performance. This is mostly due to having fewer datapoints available for accurate classification. Interestingly, we found that the improvements in coverage came from newly added clusters to the database rather than improved confidence of the Context Builder. In fact, in some cases the Context Builder became less confident, especially when having to learn to classify new events. To illustrate this, consider a context $X$ that is used to predict an event $e_{\text{old}}$. Now consider a newly observed event $e_{\text{new}}$ with the same context $X$. When the Context Builder is updated, it is taught to lower its confidence for $X$ to predict $e_{\text{old}}$ as there is now also the option to predict $e_{\text{new}}$. Nevertheless, this confidence reduction allows us to deal with new events. Moreover, the lower confidence is more than made up for by the additional coverage resulting from updating the Interpreter.

### Explainable clusters

In manual mode, DeepCASE produces clusters that security operators inspect and classify. We have shown in Section 3.5.4 that it is often enough for operators to sample a few sequences from each cluster to determine its risk level. In this section, we show what clusters look like and evaluate to what extent our attention query improves the explainability of event sequences.

**Cluster examples.** We recall that the Interpreter produces clusters by comparing the contextual events in a sequence weighted by the attention vector. These

Table 3.10: **Comparison between updating strategies.** We show the increased number of covered events when DeepCASE is updated and its performance metrics. We compare DeepCASE over time when not updated with daily, weekly and monthly updating.

| | | **Metrics over covered data** | | | |
|---|---|---|---|---|---|
| **Updates** | **Coverage** | **Precision** | **Recall** | **F1-score** | **Accuracy** |
| **None** | 87.38% | 96.19% | 92.75% | 93.67% | 92.75% |
| **Monthly** | (+411K) 93.29% | 95.64% | 92.16% | 93.16% | 92.16% |
| **Weekly** | (+450K) 93.93% | 94.84% | 90.92% | 92.30% | 90.92% |
| **Daily** | (+466K) 94.23% | 95.12% | 91.39% | 92.68% | 91.39% |

models describe the total attention of each security event type in contextual event sequences. As clusters contain similar sequences, we can describe its characteristics from the vectors detailing the total attention per event. We describe a cluster by simply averaging the attention values for each event over all sequences.

These cluster descriptions uncover interesting patterns that illustrate how an operator could reason about assigning risk levels. Figure 3.6 gives examples of these descriptions. Here the relevance of events found in the security context of a cluster is scored according to its average attention value. Consider the event in the Lastline dataset where a detector observes an `unusual user agent string`. This may be due to a newly installed or otherwise benign program, or it may be triggered by malware that does not imitate common user-agents. This event occurs 4.6K times in the dataset, which our Interpreter groups into 19 different clusters. To determine whether an event of type `unusual user agent string` is malicious or not, we have to look at the context weighted by the attention values. The first example in Figure 3.6 shows a case with 1) `NO CONTEXT` description if the event occurs where fewer than 10 detectors were triggered in the day before observing the event, indicating that there is no other supporting evidence for an attack; and 2) other detectors for the similar patterns such as `unusual JA3 fingerprint`. This cluster does not directly indicate malicious behavior. However, other cluster descriptions in Figure 3.6 show the `unusual user agent string` event in combination with detectors for the `Tor browser`. Depending on the organization, this may be considered a policy violation and can be classified as such. Finally, there are clusters with more malicious indicators such as observing `unusual user agent string` in combination with a `large data download`, or signature hits for malware (e.g., `Linkury`) or `crypto miners`.

As another example, consider the contexts observed for `beaconing activity`, which can be triggered by malware periodically contacting its command and con-

| | |
|---|---|
| `unusual user agent string` | **[benign]** |
| `NO CONTEXT` | 47.95% |
| `unusual user agent string` | 7.91% |
| `unusual JA3 fingerprint` | 42.70% |
| `unusual user agent string` | **[malicious]** |
| `NO CONTEXT` | 20.84% |
| `unusual user agent string` | 35.09% |
| `unknown crypto miner` | 43.93% |
| `beaconing activity` | **[malicious]** |
| `NO CONTEXT` | 3.68% |
| `unusual data upload` | 32.29% |
| `active directory trust enumeration` | 1.21% |
| `recently registered domain access` | 62.64% |

Figure 3.6: **Cluster description examples.** Three examples of both benign and malicious clusters. All clusters are described by the average relevance of other contextual events as described by the attention.

trol server, or can be triggered by benign software periodically checking for updates. We observe several clusters for `beaconing activity`. Some clusters contain only events by detectors looking for repetitive network connections, which do not necessarily indicate malicious activity. Other clusters, such as shown in Figure 3.6, show signs of malware because `beaconing activity` is detected in combination with `recently registered domain access` and `unusual data uploads`.

**Attention query.** One way in which DeepCASE improves its explanation of security events is through the attention query. This query increases the confidence of the Context Builder in the actual security event that occurred by shifting the attention to the more relevant contextual events. In this experiment, we measured the increase in confidence for the actual event using this attention query. Again, we trained the Context Builder on the first month of data from the Lastline dataset as in all other experiments. Next, we predicted security events from their context and measured the confidence level in the actual event that occurred with and without applying the attention query. Without the attention query, the Context Builder reached the confidence threshold of 0.2 in 86.11% of cases. Conversely, applying the attention query resulted in a confidence ≥ 0.2 in 92.21% of cases. This shows that the attention query improves the coverage for explainable events by 6.10%. For the HDFS dataset, this number increased from 94.66% to 99.24%, an improvement of 4.58 percentage points.

### 3.5.6 Robustness

Using DeepCASE to reduce the workload of security operators may also create an additional attack surface for adversaries. After all, if our approach allows adversaries to maliciously craft attacks such that DeepCASE discards them as being completely benign, reducing the workload would not serve its purpose. Moreover, our approach itself may be targeted by an attacker to annul the workload reduction for security operators. As our approach relies on events produced by security event detectors, we consider attacks that bypass or alter detector outputs to be outside the scope of this research. Therefore, we discuss and evaluate to what extent an adversary can manipulate DeepCASE itself.

#### Denial of service attack

First, an attacker can perform a denial of service (DoS) attack against DeepCASE. Here, we do not focus on (D)DoS attacks against a monitored device, as this type of attack will simply generate a single cluster. Instead, we discuss DoS attacks with the purpose of letting DeepCASE generate so many new clusters or sequences that cannot be handled by our approach, overloading security operators. In Section 3.5.7, we show that our approach is able to process more than 10 K sequences per second on a system that has a good graphics card. Therefore, we can safely assume that the bottleneck for DoS attacks is the number of sequences that the human operator has to manually inspect. While DeepCASE currently does not provide countermeasures for this type of attack, a sudden surge of new sequences to inspect would cause suspicion for a security operator, even if the individual sequences do not seem malicious. In fact, we found that in the Lastline dataset, for each device monitored in semi-automatic mode, DeepCASE triggers a new sequence to inspect only once every 2.5 days. A single device producing many more events than the average (in the Lastline dataset, the worst infected machine produced 607 unknown sequences in a single day) is likely to be thoroughly scrutinized by security operators. We discuss DeepCASE's limitations with respect to DoS attacks in Section 3.6.

#### Evasion attack

Second, an attacker may purposely trigger additional security events to change the context of the attack events. Thereby, the attacker either 1) changes the context sequence of an ongoing attack such that it matches a benign cluster instead of a malicious cluster; or 2) attempts to create a new and benign cluster that can later be used to perform an attack.

### Detecting evasive attacks

DeepCASE processes event sequences of an evading attacker in one of three ways: 1) the attack is still classified as an attack; 2) the attack is classified as benign (either by hiding in a benign cluster in manual mode, or by being assigned an INFO risk cluster in semi-automatic mode); or 3) the attack is considered an outlier and passed to a security operator. We consider case 2 the only successful outcome for an attacker, as both cases 1 and 3 will be shown to a security operator. In case an attacker tries to create a new benign cluster, such a new cluster will always be passed to a security operator, as non-matching contextual sequences are labeled as outliers. Therefore, in this experiment we evaluate to what extent an attacker can change the context of an event from a malicious to a benign cluster without becoming an outlier, which would be shown to the security operator.

### Evaluation

We simulated an evasion attack by inserting random security events in the context of events from the Lastline dataset. We note that an attacker with insider knowledge of DeepCASE can perform an evasion attack by *selecting* specific security events that perform better than random events. We discuss this scenario in Section 3.6, where we show that having specific knowledge of DeepCASE's clusters is better than inserting random events possible for only 6.32% of clusters. For the random security events, we looked only at event sequences with a risk level of LOW or higher, i.e., INFO risk levels were omitted as they did not contain attacks. In this experiment, we inserted random events into the context ranging from 0% to 100% percent of the context size in steps of 10%. Where 0% is the original context and injecting 100% completely altered the context. Next, we measured the number of sequences marked as outliers and the performance metrics over the remaining sequences.

Table 3.11 shows that the performance of DeepCASE on non-outlier sequences stays roughly the same across different numbers of injected events. However, the number of outlier sequences increases with the number of injected events. For minor perturbations, the Context Builder can detect sufficient relevant context events to be able to accurately model the sequence. However, when an attacker injects many events, the number of outlier sequences rises quickly. This means that DeepCASE notices unusual patterns and escalates these sequences to a security operator.

## 3.5.7 Runtime analysis

We evaluate the average of 10 runs of DeepCASE for various numbers of sequences. All experiments ran on a Intel(R) Xeon(R) Gold 6252 CPU @ 2.10GHz machine running Ubuntu 18.04 LTS. Neural network training and prediction ran on a NVIDIA TITAN RTX 24 GB TU102 graphics card.

Table 3.11: **Performance of DeepCASE under evasion attack.** DeepCASE becomes less confident about event sequences and sends these outlier sequences to security operators. The remaining sequences, for which DeepCASE remains confident, perform similarly to when no injection takes place.

| | | **Metrics on non-outlier data** | | | |
|---|---|---|---|---|---|
| **Injected** | **Outliers** | **Precision** | **Recall** | **F1-score** | **Accuracy** |
| 0/10 | 19.92% | 98.62% | 96.43% | 96.88% | 96.43% |
| 1/10 | 21.22% | 98.88% | 97.26% | 97.69% | 97.26% |
| 2/10 | 22.86% | 99.06% | 97.77% | 98.17% | 97.77% |
| 3/10 | 25.51% | 99.19% | 98.14% | 98.51% | 98.14% |
| 4/10 | 29.67% | 99.29% | 98.40% | 98.75% | 98.40% |
| 5/10 | 35.55% | 99.39% | 98.61% | 98.94% | 98.61% |
| 6/10 | 42.66% | 99.46% | 98.78% | 99.08% | 98.78% |
| 7/10 | 50.26% | 99.52% | 98.92% | 99.20% | 98.92% |
| 8/10 | 57.88% | 99.57% | 99.05% | 99.30% | 99.05% |
| 9/10 | 68.94% | 99.57% | 99.16% | 99.36% | 99.16% |
| 10/10 | 98.01% | 97.92% | 96.03% | 96.50% | 96.03% |

Figure 3.7 shows the result of this analysis. The total runtime is made up of four main computations: 1) training the Context Builder; 2) the attention query; 3) Interpreter's clustering in manual mode; and 4) matching known clusters in semi-automatic mode. We note that Figure 3.7 shows only a single training epoch of the Context Builder, which in reality is trained with 100 epochs. However, these epochs scale linearly and need to be trained only once for the manual mode, or when updated. From this figure, we find that it takes DeepCASE less than 5 minutes to process 1 year of data for a single company (roughly 1.2M event sequences, based on the average number of sequences in the Lastline dataset). Training the neural network of Context Builder consumes the largest part of the total runtime. We note that training epochs together with the attention query can be highly parallelized as they are performed on a GPU. Furthermore, we note that while DBSCAN clustering has a worst case complexity of $O(n^2)$, using KD-trees [39] allows us to reduce that to a complexity of $O(n \log n)$. Overall, DeepCASE can easily keep up with the number of generated events in large, real-world environments.

## 3.6 Discussion

We have shown that our approach successfully reduces the number of events presented to security operators by 95.69% in manual mode and 90.53% in semi-automatic

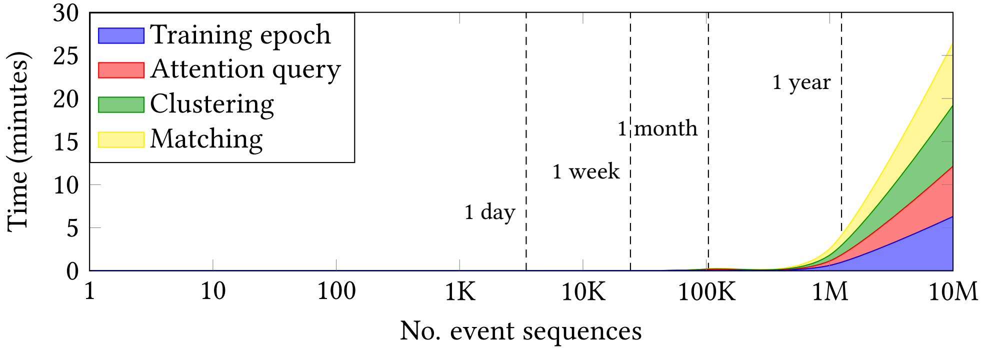


Figure 3.7: **Runtime analysis of DeepCASE.** Average runtime of our approach for different numbers of contextual sequence inputs. The runtimes of each subcomputation are stacked to show total runtime of DeepCASE. For reference, we show the number of event sequences an average organization in the Lastline dataset produces during one day, week, month and year.

mode. Nevertheless, there are some aspects of our approach to be addressed in future work.

### Bro/Zeek and programmable rules

Modern NSM and IDS systems such as Zeek [161] often include ways to manually define expert rules. As we have shown in the evaluation, such rule based systems work very well for the scenarios that they cover, and even outperform DeepCASE in terms of accuracy for the covered events. However, the main issue is that manually defined rules often have much difficulty covering all generated events. Therefore, we believe that DeepCASE offers a complementary solution to existing rule-based systems, as we ensure many more events are covered, while remaining conservative in our prediction such that less than 0.001% of events are misclassified with a lower risk level.

### Evasion

For DoS attacks against DeepCASE, the main limitation of our approach lies in the case where the attacker gradually increases the number of sequences that DeepCASE sends to a security operator. While such an attack can significantly impact the workload reduction achieved by our approach, DeepCASE will still show these sequences produced by the DoS attack to security operators. Under such an attack, DeepCASE performs equivalently to the regular setting where DeepCASE is not used.

Furthermore, we note that in our evasion experiment from Section 3.5.6 we injected random events. In reality, an adversary with knowledge of existing clusters can maliciously craft its injected events to remain undetected. We observe that the only way in which it is *possible* to change the malicious context of a malicious event *e*, is if there exist both a malicious cluster (malicious context + event *e*) *and* a benign cluster (benign context + event *e*) for that given event. We analyzed our clusters and found that only 6.32% of clusters have this property, meaning that for 93.68% of clusters, it is impossible to inject security events that would turn a malicious sequence into a benign sequence. Notably, we observed that the clusters for which switching is possible are often only a single step in a larger attack (e.g., the clusters for `beaconing activity`, where detection of connecting to the command and control server can be circumvented). Other steps of the attack, such as uploading large amounts of data or signature hits are more difficult to bypass without being triggered as outliers. In short, evasion attacks may be possible for 6.32% of the clusters if an attacker exploits knowledge of how clusters are formed, which is a limitation of DeepCASE.

### SOAR systems

Our work shows that DeepCASE creates clusters of similar event sequences that correspond to similar risk levels for a machine. However, these clusters are used only to filter benign sequences and classify the risk levels of malicious sequences before showing them to security operators. Similar event sequences intuitively signal similar threats infecting a device. This would mean that the response of the security operator is also similar in terms of removing the threat and patching the device. Such automated response systems are known in industry as “security orchestration, automation and response”, i.e., SOAR systems. In its current form DeepCASE could help SOAR systems when operators create automated responses per cluster.

### Relative risk levels

In its current form, DeepCASE does not distinguish between risk levels for different machines or organizations, it merely produces a risk analysis based on previously observed event sequences. However, in practice, devices that are vital for an organization may have a lower tolerance for potential risks than other devices. As our approach analyzes event sequences on a per-device basis, we suggest to forward lower risk event sequences to the security operators for vital devices. If this is done only for a small set of devices, the impact on workload reduction should be minimized. However, further research is required to evaluate the full impact on workload reduction.

### Attention and explainability

Our approach uses an attention mechanism to select relevant parts of the context of an event. This attention mechanism is a popular approach in the current state-of-the-art research into natural language processing (NLP). In this domain, there is an ongoing discussion whether attention may be used for explaining feature importance [110]. The main critique here is that the attention vector used in state-of-the-art works (e.g., BERT [72]) does not apply attention 1-to-1 to each input, but rather maps attention to a complex combination of different inputs. We mitigate this critique by multiplying the attention directly with the embedding of each context event creating a direct mapping (see the Event Decoder in Section 3.3.2). Furthermore, the results from our evaluation show that combining the attention with each individual event can be used for accurate matching and filtering of event sequences.

### Transferability

Our approach models event sequences based directly on the events produced by underlying security detection systems. This means that changes in the detectors will affect the performance of DeepCASE in terms of filtered sequences. We have shown that our approach can automatically update itself with new detectors in Section 3.5.5. In some scenarios, an operator may want to take a pre-trained model from one organization and apply it to another organization to avoid having to run the manual mode. Further research could show us how well existing models can transfer to other settings, using methods such as transfer learning [159].

### Context

One limitation of DeepCASE is that it only deals with events in the context of the same machine. While it would be interesting to find cross-host relations between events, we view this as future work. Another limitation of DeepCASE is the limited size of the context it can deal with. Therefore, attacks over long periods of time and contexts filled up with many unrelated events cannot be properly assessed. Our parameter selection (Section 3.5.2) showed that incrementing the size of the context (both in terms of time and number of events) beyond 5 samples and 1 hour has limited effects on the performance.

### Transformers

Our approach uses an Attention-based Encoder-Decoder model. Recent advances in the field of NLP have improved this type of architecture in the form of Transformers [210], of which notable examples are BERT [72] and GPT-3 [46]. These transformers are based on the same concept of attention as our work, but offer a larger amount of parallelization. For a more detailed description, we refer to the original paper [210]. While this transformer architecture would also work for DeepCASE, the increased complexity of such a network would add little helpful insights into our main concept, namely that attention can be used to explain relations between security events.

## 3.7 Related work

Related works have explored various ways to contextualize and predict security events, automating the operators' tasks.

### 3.7.1 Contextual security events

The default method of analysing security events is provided by expert rules such as those provided by AlienVault's OSSIM, Sigma, and the rules that can be programmed in software such as Zeek [161]. However, as we have shown in the evaluation, these rules often only cover a limited subset (in our evaluation a maximum of 36.97%) of all event sequences that we observe. More automated methods such as NoDoze [99] and UNICORN [98] model the security context of system-level events and organization-wide events, respectively. Both approaches model this context as provenance graphs that track which processes are connected to triggered events. They then automatically assign an anomaly score to this sequence to assist the triaging process. The main drawback of these approaches is that it requires process-level information of monitored hosts in order to construct provenance graphs. Other works, such as OmegaLog [100], go even further in providing security operators with additional provenance information by analyzing the executed binaries. Our work drops the requirement for process-level information altogether by focusing only on the security events themselves, and later identifying correlated events using the Context Builder. This allows us to handle events of less device-intrusive detection mechanisms such as network-level security detectors. Additionally, security operators can monitor more devices, including those operating in bring-your-own-device settings.

### 3.7.2 Event prediction

Other works do not focus on classifying the threat level of attacks, but rather focus on predicting the next attack steps given a sequence of prior security events.

DeepLog [76] employs a recurrent neural network to predict future events. In case the predicted event does not happen, DeepLog raises an alert for an anomaly in the event sequence. When their approach detects an anomaly, it is considered malicious and passed through a separate workflow extraction system to detect the underlying cause. Like many other systems, DeepLog focuses on prediction of system logs, which means that it is optimized for more detailed events. This can also be observed from the similarity in performance between DeepCASE and DeepLog on the HDFS dataset, and the difference in the Lastline dataset. Furthermore, DeepLog focuses only on anomalies, but an anomaly is not necessarily malicious.

Tiresias [191] does not predict anomalies, but instead tries to accurately predict future events. While achieving a decent performance with an F1-score of 95.68% versus 97.90% of DeepCASE, Tiresias is a complete black-box approach to event prediction. This means that security operators have no way of telling whether a particular prediction is meaningful given a specific context.

In another paper, the same authors propose Attack2vec [192], which detects *changes* in attack patterns based on differences in preceding security events. While this is also a form of contextual analysis, the goal and approach are strictly different. Where Attack2vec only detects changes in attack trends, DeepCASE is able to also cluster these new attacks and present them to security operators to immediately take action.

One of the earlier works in security event prediction is Nexat [57], which uses a co-occurrence matrix of security events to predict the most likely next event. However, this approach both assumes straightforward relations between events and is fully supervised. This makes it more difficult to deal with rapidly evolving attack patterns as it constantly needs to be retrained. Conversely, DeepCASE offers security operators a simple and effective approach to semi-automatically update itself to deal with novel event sequences.

### Attention-mechanisms

ALEAP [80] uses an attention mechanism for event prediction. Unfortunately, their work does not leverage this mechanism to provide contextual analysis, and, therefore, cannot properly assist security operators in their work. Moreover, their approach only has a prediction performance of only 72.36% precision (no mention of accuracy/recall/F1-score). Besides the lower performance, the complexity of the model is much higher than the one used in our approach, leading to longer training and prediction times.

Brown et al. [45] also use attention mechanisms to enhance prediction of security log messages. However, they predict specific log attributes based on different attributes within the same log message. Hence, they base their prediction of maliciousness on meta-data of individual messages, such as the machine generating an event, or the user that authenticated an action, instead of other activities within the network. This is simply a different approach to what many individual security detection mechanisms already do.

### Graph-based approaches

Other works capture context through graph-based approaches, which can be built from data [100, 133, 193]. The disadvantage of such approaches is that they rely on predefined rules to model context. Therefore, new patterns will remain unobserved, making it difficult to give a complete overview of the context.

## 3.8 Conclusion

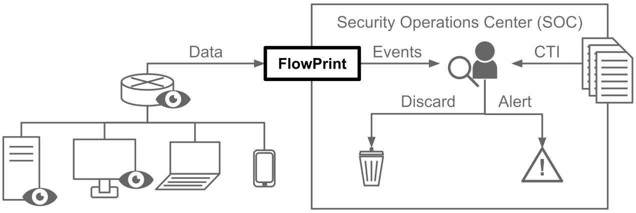
In this chapter, we proposed DeepCASE, a novel approach that assists security operators in analyzing security events by inspecting the context of events. Unlike existing approaches, our approach does not require system-level information and can therefore be used to analyze security events of any type of security detector. Moreover, we showed that DeepCASE is able to deal with complex and evolving attacks without resorting to a black-box approach.

Additionally, we showed that DeepCASE reduces the workload of security operators on real-world data by 95.39%, and semi-automatically handles 90.53% of events with an accuracy of 94.34%. Moreover, DeepCASE underestimates risk in less than 0.001% of cases, showing that real attacks are rarely missed. These results demonstrate that contextual event analysis is an effective technique for security event analysis and a useful tool for real-world security operations centers.

# Chapter 4

## FlowPrint: Semi-Supervised Mobile-App Fingerprinting on Encrypted Network Traffic

While triaging security events reduces the workload of security operators, it assumes that events provide sufficient contextual information for analysis. Where EDR can provide information on the application exhibiting suspicious behavior, network-based solutions often cannot. Therefore, in this chapter, we answer the following research question:



**RQ 3.** *To what extent can we identify malicious applications based on network traffic?*

Mobile-application fingerprinting of network traffic is valuable for many security solutions as it provides insights into the apps active on a network. Unfortunately, existing techniques require prior knowledge of apps to be able to recognize them. However, mobile environments are constantly evolving, i.e., apps are regularly installed, updated, and uninstalled. Therefore, it is infeasible for existing fingerprinting approaches to cover all apps that may appear on a network. Moreover, most mobile traffic is encrypted, shows similarities with other apps, e.g., due to common libraries or the use of content delivery networks, and depends on user input, further complicating the fingerprinting process.

Therefore, we propose FlowPrint, a semi-supervised approach for fingerprinting mobile apps from (encrypted) network traffic. We automatically find temporal correlations among destination-related features of network traffic and use these correlations to generate app fingerprints. Our approach is able to fingerprint previously unseen apps, something that existing techniques fail to achieve. We evaluate our approach for both Android and iOS in the setting of app recognition, where we achieve an accuracy of 89.2%, significantly outperforming state-of-the-art solutions. In addition, we show that our approach can detect previously unseen apps with a precision of 93.5%, detecting 72.3% of apps within the first five minutes of communication.

This chapter is based on the publication: [3] Thijs van Ede et al. "Flowprint: Semi-supervised mobile-app fingerprinting on encrypted network traffic". In: *Network and Distributed System Security Symposium (NDSS)*. vol. 27. 2020

## 4.1 Introduction

Security solutions aim at preventing potentially harmful or vulnerable applications from damaging the IT infrastructure or leaking confidential information. In large enterprise networks, this is traditionally achieved by installing monitoring agents that protect each individual device [214]. However, for mobile devices security operators do not have direct control over the apps installed on each device in their infrastructure, especially when new devices enter networks under bring-your-own-device (BYOD) policies on a regular basis and with the ease by which apps are installed, updated, and uninstalled. In order to still retain detection capabilities for apps that are active in the network, operators rely on observing the network traffic of mobile devices. This naturally introduces the challenge of detecting apps in *encrypted* network traffic, which represents the majority of mobile traffic—80% of all Android apps, and 90% of apps targeting Android 9 or higher, adopt Transport Layer Security (TLS) [91].

However, recognizing mobile apps can be a double-edged sword: On the one hand, network flow analysis provides a non-intrusive central view of apps on the network without requiring host access. On the other hand, app detection can be used for censoring and invades users' privacy. As we show in this chapter, active apps on a network can not only be reliably fingerprinted for security purposes, but also in an adversarial setting, despite traffic encryption. Thus, privacy-conscious users need to be aware of the amount of information that encrypted traffic is still revealing about their app usage, and should consider additional safeguards, such as VPNs, in certain settings.

The idea of network-based app detection has already been extensively explored in both industry and academia [13, 16, 52, 65, 149]. Snort for example offers AppID [59], a system for creating network intrusion detection rules for specified apps, while Andromaly [189] attempts to detect unknown software through anomaly detection by comparing its network behavior to that of known apps. Other approaches specifically focus on detecting apps containing known vulnerabilities [203], and others identify devices across networks based on the list of apps installed on a device [198]. All these approaches have in common that they require prior knowledge of apps before being able to distinguish them. However, new apps are easily installed, updated and uninstalled, with almost 2.5 million apps to choose from in the Google Play Store alone [197], not to mention a number of alternative markets. Furthermore, recent work has shown that even the set of pre-installed apps on Android varies greatly per device [87]. Thus, especially when companies adopt BYOD policies, it is infeasible to know in advance which apps will appear on the network. As a consequence, unknown apps are either misclassified or bundled into a big class of unknown apps. In a real-world setting, a security operator would need to inspect the unknown traffic and decide which app it belongs to, limiting the applicability of existing approaches in practice.

Unlike existing solutions, we assume *no prior knowledge* about the apps running in the network. We aim at generating fingerprints that act as markers, and that can be used to both recognize known apps and automatically detect and isolate previously unseen apps. From this, a security operator can update whitelists, blacklists or conduct targeted investigations on per-app groupings of network traffic.

There are several challenges that make such fingerprinting non-trivial. This is because mobile network traffic is particularly *homogeneous*, highly *dynamic*, and constantly *evolving*:

**Homogeneous.** Mobile network traffic is *homogeneous* because many apps share common libraries for authentication, advertisements or analytics [32]. In addition, the vast majority of traffic uses the same application-level protocol HTTP in various forms (HTTP(S)/QUIC) [174]. Furthermore, part of the content is often served through content delivery networks (CDNs) or hosted by cloud providers. Consequently, different apps share many network traffic characteristics. Our work tackles homogeneous traffic by leveraging the difference in network destinations that apps communicate with. We show that despite the large overlap in destinations, our approach is still able to extract unique patterns in the network traffic.

**Dynamic.** Mobile network traffic is often *dynamic* as data that apps generate may depend on the behavior of the users, e.g., their navigation through an app. Such dynamism may already be observed in synthetic datasets that randomly browse through an app's functionality. Various fingerprinting approaches rely on repetitive behavior in network traffic [12, 83]. Despite good results of these methods in smart-home environments and industrial control systems, dynamic traffic could complicate fingerprinting of mobile apps. Hence, our work aims to create fingerprints that are robust against user interactions by leveraging information about network destinations on which the user has limited influence. We show that our approach achieves similar results on both automated and user-generated datasets.

**Evolving.** Mobile network traffic is constantly *evolving* as app markets offer effortless installation, update, and uninstallation of a vast array of apps. Studies have shown that apps are regularly updated with new versions, as frequently as once a month on average [32, 60]. This is a challenge for existing fingerprinting mechanisms that require prior knowledge of an app in order to generate fingerprints. When new or updated apps are introduced into the network, these fingerprinting systems become less accurate, similarly to what Vastel et al. observed in the setting of browser fingerprinting [209]. Moreover, the fraction of apps covered by these systems dramatically decreases over time if fingerprints are not regularly updated.

Our solution counters this by basing its fingerprints on pattern discovery in network traffic instead of training on labeled data. Doing so, our approach produces fingerprints that automatically evolve together with the changing network traffic. We show that our approach is able to correctly recognize updated apps and can even detect and fingerprint previously unseen apps.

To address these challenges, we introduce a semi-supervised approach to generate fingerprints for mobile apps. Our key observation is that mobile apps are composed of different modules that often communicate with a static set of destinations. We leverage this property to discover patterns in the network traffic corresponding to these different modules. On a high level, we group together (encrypted) TCP/UDP flows based on their destination and find correlations in destinations frequently accessed together. We then combine these patterns into fingerprints, which may, among other use cases, be used for app recognition and unseen app detection.

While our approach does not require prior knowledge to generate fingerprints, and could, thus, be considered unsupervised, the applications of our approach are semi-supervised. In fact, our approach creates "anonymous" labels that uniquely identify mobile apps. However, app recognition uses known labels to assign app names to the matched fingerprints. For example, having knowledge about the Google Maps app, allows us to rename `unknown_app_X` to `google_maps`. Similarly, unseen app detection requires a training phase on a set of known apps to identify unknown ones.

In summary, we make the following contributions:

- We introduce an approach for semi-supervised fingerprinting by combining destination-based clustering, browser isolation and pattern recognition.
- We implement this approach in our prototype FlowPrint, the first real-time system for constructing mobile app fingerprints capable of dealing with unseen apps, without requiring prior knowledge.
- We show that, for both Android and iOS apps, our approach detects known apps with an accuracy of 89.2%, significantly outperforming the state-of-the-art supervised app recognition system AppScanner [203]. Moreover, our approach is able to deal with app updates and is capable of detecting previously unseen apps with a precision of 93.5%.

In the spirit of open science, we make both our prototype and datasets available at `https://doi.org/10.4121/e08823b5-ceff-4ebc-a967-290ef9cacc7e`.

## 4.2 Preliminary Analysis

To study mobile network traffic and identify strong indicators that can be used to recognize mobile apps, we performed a preliminary analysis on a small dataset.

As indicated in the introduction, our fingerprinting method should be able to distinguish mobile apps despite their homogeneous, dynamic and evolving behavior. Hence, in our preliminary analysis we explored features that may be used to fingerprint apps.

### 4.2.1 Dataset

In order to perform our analyses, we use datasets of encrypted network traffic labeled per app (see Table 4.1). These datasets allow us to evaluate our method in various conditions as they contain a mix of both synthetic and user-generated data; Android and iOS apps; benign and potentially harmful apps; different app stores; and different versions of the same app. We collected three of the datasets as part of our prior work [132, 176–178]. We collected the last set specifically for this work with the purpose of representing browser traffic, which is lacking in most available datasets. For this preliminary analysis, we only used a small fraction of the available data in order to prevent bias in the final evaluation.

**ReCon.** The ReCon AppVersions dataset [177, 178] consists of labeled network traces of 512 Android apps from the Google Play Store, including multiple version releases over a period of eight years. The traces were generated through a combination of automated and scripted interactions on five different Android devices. The apps were chosen among the 600 most popular free apps on the Google Play Store ranking within the top 50 in each category. In addition, this dataset contains extended traces of five apps, including multiple version releases. The network traffic of each of these five apps was captured daily over a two-week period. In this work, we refer the AppVersions dataset as *ReCon* and to the extended dataset as *ReCon extended.*

**Cross Platform.** The *Cross Platform* dataset [176] consists of user-generated data for 215 Android and 196 iOS apps. The iOS apps were gathered from the top 100 apps in the App Store in the US, China and India. The Android apps originate from the top 100 apps in Google Play Store in the US and India, plus from the top 100 apps of the Tencent MyApps and 360 Mobile Assistant stores, as Google Play is not available in China. Each app was executed between three and ten minutes while receiving real user inputs. Procedures to install, interact, and uninstall the apps were given to student researchers who followed them to complete the experiments while collecting data. We use this dataset to evaluate both the performance of our method with user-generated data and the performance between different operating systems.

Table 4.1: **Dataset overview.** ✓ indicates a dataset contains Homogeneous (H), Dynamic (D), or Evolving (E) traffic.

| Dataset | No. Apps | No. Flows | % TLS Flows | Start Date | End Date | Avg. Duration | H | D | E |
|---|---|---|---|---|---|---|---|---|---|
| ReCon [177, 178] | 512 | 28.7K | 65.9% | 2017-01-24 | 2017-05-06 | 189.2s | ✓ | | ✓ |
| ReCon extended [177, 178] | 5 | 141.2K | 54.0% | 2017-04-21 | 2017-05-06 | 4h 16m | ✓ | | ✓ |
| Cross Platform (Android) [176] | 215 | 67.4K | 35.6% | 2017-09-11 | 2017-11-20 | 333.0s | ✓ | ✓ | |
| Cross Platform (iOS) [176] | 196 | 34.8K | 74.2% | 2017-08-28 | 2017-11-13 | 339.4s | ✓ | ✓ | |
| Cross Platform (All) [176] | 411 | 102.2K | 49.6% | 2017-08-28 | 2017-11-20 | 336.0s | ✓ | ✓ | |
| Andrubis [132] | 1.03M | 41.3M | 24.7% | 2012-06-13 | 2016-03-25 | 210.7s | ✓ | | |
| Browser | 4 | 204.5K | 90.5% | 2018-12-17 | 2019-03-01 | 3h 34m | ✓ | | |

**Andrubis.** The *Andrubis* dataset [132] contains labeled data of 1.03 million Android apps from the Google Play Store and 15 alternative market places. This dataset contains both benign and potentially harmful apps, as classified by VirusTotal. Each trace in this dataset was generated by running the app for four minutes in a sandbox environment emulating an Android device. The app was exercised by automatically invoking app activities and registered broadcast receivers, and simulating user interactions through the Android Application Exerciser Monkey. We use the Andrubis dataset for experiments requiring large traffic volume and to assess the performance of our method on both benign and potentially harmful apps.

**Browser.** We created the *Browser* dataset because the existing datasets contain a limited amount of browser traffic, which may produce a significant portion of traffic in mobile environments. Even though a browser is not a dedicated app, but rather a platform on which various web content is rendered, executed and displayed, a fingerprinting method with the purpose of detecting apps should also be able to detect the browser as a single app. To this end, we collect an additional dataset of browser traffic by scraping the top 1,000 Alexa websites on a Samsung Galaxy Note 4 running Android 6.0.1 with Chrome, Firefox, Samsung Internet and UC Browser, which cover 90.9% of browser traffic [196], if we exclude Safari, which is not available for Android. Each website visit lasts for 15 seconds, while the Application Exerciser Monkey simulates a series of random movements and touches.

### 4.2.2 Feature Exploration

Previous work on app fingerprinting usually tackles the problem in a supervised setting. In this work however, we propose an approach with the aim of automatically detecting unknown apps, without requiring prior knowledge. This requires a re-evaluation of the network features commonly used in app fingerprinting. Therefore, we first identify possible features from the network traffic. The TLS-encrypted traffic limits the available features to temporal and size-based features, as well as the header values of unencrypted layers and the handshake performed to establish a TLS connection. The data-link layer header provides only information about the linked devices, not about the app itself and is therefore not useful for our purposes. We further analyze the layers between the data-link and application layer, as we expect the latter to be encrypted. From these layers, we extract all header values controlled by the communicating app as well as the sizes and inter-arrival times of packets. In addition, for the size and time related features we compute the statistical properties: minimum, maximum, mean, standard deviation, mean absolute deviation, and 10-th through 90-th percentile values.

### 4.2.3 Feature Ranking

We score all features according to the Adjusted Mutual Information (AMI) [211], a metric for scoring features in unsupervised learning. We favor the AMI over other methods, such as information gain, as the latter is biased towards features that can take on random values. Such randomness is undesirable in an unsupervised or semi-supervised setting, as we do not have any prior expectation of feature values. The AMI defines the relative amount of entropy gained by knowing a feature with respect to the class, in our case the app. To this end, we first compute the mutual information between a feature and its app as described in Equation 4.1. Here $Y$ is the list of classes of each sample and $X$ is the list of features corresponding to the samples. Function $p(x, y)$ defines the joint probability of value $x$ and label $y$, whereas $p(x)$ and $p(y)$ are the individual probabilities of features $x$ and $y$ occurring respectively.

$$MI(X, Y) = \sum_{y \in Y} \sum_{x \in X} p(x, y) \log\left(\frac{p(x, y)}{p(x)p(y)}\right) \tag{4.1}$$

To counter that the mutual information is biased toward features that have many different values, the AMI removes any bias by normalizing for the expected gain in entropy. As a result, the AMI score ranges from 0 (completely uncorrelated) to 1 (observing feature X fully maps to knowing app label Y). Equation 4.2 shows the definition of the AMI, where $E[X]$ is the expected value of $X$ and $H(X)$ is the entropy of $X$. We use the AMI to rank features based on how much information they contain about an app, and thereby get an indication of their usefulness in a fingerprint.

$$AMI(X, Y) = \frac{MI(X, Y) - E[MI(X, Y)]}{\max(H(X), H(Y)) - E[MI(X, Y)]} \tag{4.2}$$

Table 4.2: **AMI of ten highest scoring features.**

| **Feature** | **Category** | **AMI** |
|---|---|---|
| Inter-flow timing | Temporal | 0.493 |
| IP address - source | Device | 0.434 |
| TLS certificate - validity after | Destination | 0.369 |
| TLS certificate - validity before | Destination | 0.356 |
| TLS certificate - serial number | Destination | 0.342 |
| IP address - destination | Destination | 0.246 |
| TLS certificate - extension set | Destination | 0.235 |
| Packet size (incoming) - std | Size | 0.235 |
| Packet size (outgoing) - std | Size | 0.232 |
| Packet inter-arrival time (incoming) - std | Temporal | 0.218 |

### 4.2.4 Feature Evaluation

Using the AMI, we analyze and rank the features available in TLS-encrypted traffic of the ReCon dataset. The evaluation of our fingerprinting approach in Section 4.5 demonstrates that these features also generalize to other datasets. After extracting all features, we divide them into categorical and continuous values. As the AMI can be compared only for categorical values, we divided each continuous value into 20 equally sized bins spanning the full range of each feature. Then, we computed the AMI of each feature with respect to the app label. Table 4.2 shows the ten highest ranked features, we provide all the analyzed features together with their AMI scores at `https://doi.org/10.4121/e08823b5-ceff-4ebc-a967-290ef9cacc7e`.

From Table 4.2 we first observe that there are no features with an AMI close to 1. Hence, a fingerprint should combine multiple features in order to create a reliable app marker. In addition, we deduce four important categories that can be leveraged when creating app fingerprints. We note that these categories are not new to app fingerprinting, but give insights into how an approach may benefit from leveraging these features. While only using a small part of the dataset for our preliminary feature evaluation, our results in Section 4.5 show that the features are generic and also perform well on larger datasets.

**Temporal features.** The *Inter-flow timing* and *Packet inter-arrival time (incoming)* stress the importance of timing in network traffic. Most apps primarily communicate when active, and early studies suggested a limited number of apps are active simultaneously [41, 74]. As temporal features may be affected by latency and network congestion on a small-time scale, our work uses time on a more course-grained level. We leverage the time between flows to correlate traffic occurring at the same time interval. In addition to our semi-supervised setting, supervised fingerprinting methods such as BIND [149] also use temporal features.

**Device features.** The *IP address - source* feature is the IP address of the monitored device. This feature demonstrates that the device producing network traffic reveals information about the app. Intuitively, different devices may run different app suites. Our work does not use the IP source address as a feature, but instead creates app fingerprints separately per device. We reason that identifying apps on a per-device basis assists in limiting the amount of dynamic behavior. Furthermore, a related study [16] observed that different devices in terms of vendor and/or OS version may exhibit significant variations in traffic features. Therefore, our approach handles traffic on a per-device basis and constructs separate fingerprints for each device.

**Destination features.** The high AMI of the *IP address - destination*, i.e., the IP address of the server, and various *TLS certificate* features indicate that apps may be discriminated based on the destinations with which they communicate. Intuitively, each app is composed of a unique set of different modules that all provide parts of the app's functionality. Each module communicates with a set of servers resulting in a unique set of network destinations that differentiate apps. Destination features may even be enriched by domains extracted from DNS traffic. However, this data is not always available due to DNS caches. Hence, to work in a more general setting our approach does not use the domain as a feature. Even though network destinations may change over time, we show in Section 4.5 that our approach is able to deal with these changes.

**Size features.** Both incoming and outgoing *Packet size* features show a high AMI. This implies that the amount of data being sent and received per flow is a good indicator of which app is active. However, all other packet size features yielded an AMI score of 0.07 or lower, i.e., making up two thirds of the bottom 50% of ranked features. Therefore, we do not incorporate packet sizes in our approach. This does not mean size features are unsuited for fingerprinting per se, as can be observed from supervised approaches using size-based features [13, 16, 203]. However, the size features yield little information for fingerprinting in a semi-supervised setting.

## 4.3 Threat Model

Our work focuses on creating fingerprints for mobile apps and we assume the perspective of a security monitor who can (1) trace back flows to the device despite NAT or changing IP addresses, (2) distinguish mobile from non-mobile devices, and (3) only monitor its own network (e.g., the WiFi network of an enterprise)—traffic sent over other networks cannot be used to generate fingerprints. Our assumptions match the scenario of enterprise networks, where security operators have full network visibility and access to the DHCP server.

Even without a priori knowledge about device types, security operators could still isolate network traffic from mobile devices based on MAC addresses and orthogonal OS fingerprinting approaches: for example, related work has shown that DHCP messages [160], TCP/IP headers [50], and OS-specific destinations [122] (e.g., update servers and mobile app markets), can be used to identify mobile devices, and even tethering.

Finally, we focus on single-app fingerprints, i.e., we assume that mobile apps are executed one at a time. In practice, there is often a separation between the execution of multiple apps, with the exception of background services, which, however, produce fewer and more recognizable traffic patterns. Nonetheless, we acknowledge the possibility that multiple apps are executed simultaneously on a single device causing composite fingerprints. We believe our approach is an excellent start to investigate the creation and behavior of such composite fingerprints. However, as we will discuss in Section 4.6, we consider this out of scope for the current work as existing solutions already suffer from limitations such as identifying previously unseen apps.

## 4.4 Approach

We aim to fingerprint mobile apps in a semi-supervised and real-time fashion on the basis of their (encrypted) network traffic. We build our approach on the observation that mobile apps are composed of different modules that each communicate with a relatively unvarying set of network destinations. Our focus lies on discovering these distinctive communication patterns without requiring any knowledge of the specific active apps. To this end, we create fingerprints based on temporal correlations among network flows between monitored devices and the destinations they interact with. As a result, our fingerprints are capable of dealing with evolving app suites, and are agnostic to the homogeneous and dynamic nature of mobile traffic.

Figure 4.1 shows an overview of our approach: We periodically take network traffic of mobile devices as input and generate fingerprints that map to apps. To do so, we isolate TCP/UDP flows from the network traces for each device, and extract the required features. Subsequently, for each individual device we cluster all flows according to their destination. This clustering allows the discovery of common communication patterns later on. Before generating app fingerprints, our approach first pays special attention to browsers as they behave like a platform accessing web content rather than a dedicated app. Thereafter, we correlate remaining clusters based on temporally close network activity to generate app fingerprints. When clusters

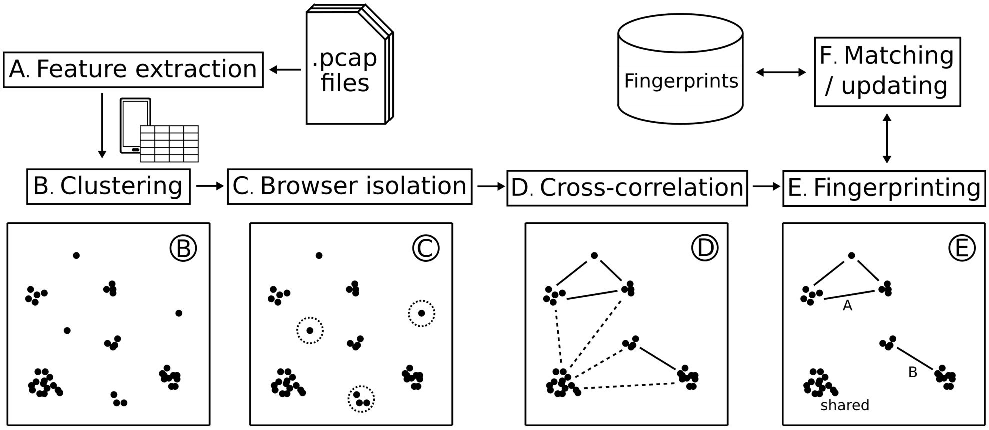


Figure 4.1: **Overview of the creation and matching of app fingerprints.** (A) We extract features from the network traces. (B) We cluster the flows from each device per network destination. (C) We detect and isolate browsers. (D) We discover correlations between network destinations. (E) We create fingerprints based on strong correlations. (F) We match newly found fingerprints against previously generated fingerprints and update them accordingly.

show a strong correlation, we group their flows together in a fingerprint. Finally, we match the generated fingerprints against a database of known fingerprints to recognize apps or detect previously unseen apps. By combining correlation and clustering techniques, our approach discovers temporal access patterns between network destinations without requiring any prior knowledge.

### 4.4.1 Feature Extraction

The first step for generating fingerprints extracts features from the network traffic, where we separately look at the TCP and UDP flows of each mobile device. Per device, we extract the destination IP and port number, timestamp (used to compute the timing between flows), size and direction of all packets in the flow and, if applicable, the TLS certificate for that flow. From these features, we use the destination IP and port number as well as the TLS certificate in the clustering phase. Browser isolation additionally requires information about the amount of data that is sent over the network. Finally, the correlation step uses the timestamps of packets to determine to what extent different flows are temporally correlated.

### 4.4.2 Clustering

Since our approach runs periodically over input data of each device, we first split the input data is into batches of a given timing interval $\tau_{\text{batch}}$. After extracting the features for each batch, we cluster together TCP/UDP flows based on their network destination. We consider flows to go to the same network destination if they satisfy any of the following criteria: (1) The flows contain the same (destination IP address, destination port)-tuple. (2) The flows contain the same TLS certificate.

The clustering approach for app fingerprinting raises some concerns about the consistency of destination clusters. After all, web services may use multiple IP addresses for a single destination for load balancing and reducing the server response time, or even change their IP address completely. Our approach tackles this problem by clustering destinations based on similarity of either the (IP, port)-tuple or the TLS certificate. As discussed previously, one may even enrich the clustering features by including DNS traffic of flows as well if this information is available. Our evaluation in Section 4.5 shows that this method is robust against inconsistencies in network destinations.

Figure 4.2 shows an example of the resulting clusters, in which the destination clusters are scattered randomly. The size of each cluster is proportionate to the amount of flows assigned to it. Note that some of the clusters are generated by multiple apps, which we refer to as *shared clusters*. Further inspection reveals that these shared clusters correspond to third-party services such as crash analytics, mobile advertisement (ad) networks, social networks, and CDNs. These services are often embedded through libraries used by many apps: e.g., `googleads.g.doubleclick.net`, `lh4.googleusercontent.com` and `android.clients.google.com` are shared clusters that provide these services. We discuss the extent to which shared clusters influence fingerprinting in our analysis on homogeneity in Section 4.5.5. In addition to shared clusters, apps frequently produce clusters unique to that specific app, e.g., the `s.yimg.com` and `infoc2.duba.net` clusters only occur in the traffic of the *com.rhmsoft.fm* app. These app-specific clusters often point to destinations of the app developer, i.e., the first party, or smaller providers of the aforementioned cross-app services.

Finally, note that the obtained clusters consist of flows from the entire input batch. However, the monitored device will only sporadically communicate with each destination. Therefore, we refer to clusters as *active* when a message is sent to or received from the destination represented by the cluster, and *inactive* otherwise.

### 4.4.3 Browser Isolation

As previously discussed, browsers are different from other apps in that they are not dedicated apps. This means that behavioral patterns in browsers are more difficult to detect as the user may navigate to any website at will. To account for this, we introduce a separate technique to detect and isolate browser traffic into a single app.

**Features.** From the perspective of destination clustering, we expect browsers to show many new clusters. After all, modern websites distribute their content along CDNs, display advertisement, and load auxiliary scripts and images. These are stored at various destinations and therefore show up as new clusters. In addition, content downloaded to be displayed in browsers often contains much more data than is uploaded in browser requests. To account for the fact that multiple apps may be active and thereby show browser-like behavior, we focus only on the relative changes. Therefore, our browser detector uses the following features: (1) Relative change in active clusters; (2) Relative change in bytes uploaded; (3) Relative change in bytes downloaded; (4) Relative change in upload/download ratio.

**Browser detector.** To detect browsers, we train a Random Forest Classifier [102] with labeled browser and non-browser data.[1] When the classifier detects a TCP/UDP stream originating from a browser at time $t$, we isolate all connections active within an empirically set timeframe of $[t-10, t+10]$ seconds. This means that we label the connections as *browser* and do not consider them for further analysis. Therefore, after detection, these streams are removed from the destination clusters. Our rationale for removing all connections within a specific timeframe is that, when a browser is detected, it probably caused more network activity around that time. While this approach might be considered aggressive in detecting browsers, we argue that other apps should show persistent behavior. As a result, clusters that have been removed because all their connections were incorrectly isolated are still expected to resurface when the app is active without an interfering browser. We evaluate the performance of the browser isolation component in Section 4.5.4.

### 4.4.4 Cluster Correlation

Now that browsers are isolated, we leverage the remaining clusters for app fingerprinting. However, using only destination clusters is insufficient for fingerprinting apps as network destinations are shared among apps and may change between different executions of an app [202, 203]. A small-scale experiment on our datasets shows that an increasing number of apps leads to a rapid decline in app-specific clusters. When randomly selecting 100 apps from all our dataset over ten Monte Carlo cross validations, only 58% of apps show at least one app-specific destination cluster. In the same experiment, when selecting 1,000 apps, this number drops to 38%. Therefore, to fingerprint apps we also leverage the temporal correlations between active destination clusters. Our rationale here is that apps persistently communicate with the same network destinations. We hypothesize that the combination of active destination clusters at each point in time is unique and relatively stable for

[1] While this is a form of supervised detection, we still consider our approach semi-supervised as we do not require prior knowledge for other types of apps.

each app. This means that over time one should be able to observe stronger correlations for destinations that belong to the same app. Our experiments in Section 4.5 demonstrate that this method of fingerprint generation can be used for both app recognition and detection of previously unseen apps.

**Correlation graph.** To measure the temporal correlation between clusters, we compute the cross-correlation [169] between the activity of all cluster pairs as defined in Equation 4.3. Even though this has a theoretical time complexity of $O(n^2)$, we show in Section 4.5.7 that in practice it is still easily scalable. We compute this cross-correlation by splitting the input batch into slices of $\tau_{\text{window}}$ seconds (see Section 4.5.1). We consider a cluster $c_i$ active at time slice $t$ if it sends or receives at least one message to or from the destination cluster during that window. Its activity is modeled as $c_i[t] = 1$ if it is active or $c_i[t] = 0$ if it is inactive.

$$(c_i \star c_j) = \sum_{t=0}^{T} c_i[t] \cdot c_j[t] \tag{4.3}$$

The cross-correlation is naturally higher for clusters with a lot of activity. To counter this, we normalize the cross-correlation for the total amount of activity in both clusters as specified in Equation 4.4.

$$(c_i \star c_j)_{norm} = \frac{\sum_{t=0}^{T} c_i[t] \cdot c_j[t]}{\sum_{t=0}^{T} \max(c_i[t], c_j[t])} \tag{4.4}$$

Using the cross-correlation metric between each cluster, we construct a correlation graph with each node in this graph representing a cluster. Clusters are connected through weighted edges where the weight of each edge defines the cross-correlation between two clusters. Figure 4.3 shows the correlation graph of three selected apps as an example. We see that clusters belonging to the same app demonstrate a strong cross-correlation. In addition, shared clusters show weak correlation between all apps and most of the unique clusters are not correlated at all.

### 4.4.5 App Fingerprints

To construct app fingerprints we identify maximal cliques, i.e., complete subgraphs, of strongly correlated clusters in the correlation graph. To discover such cliques, we first remove all edges with a weak cross-correlation. A cross-correlation is considered weak if it is lower than a threshold $\tau_{\text{correlation}}$, which in our approach is empirically set to 0.1 (see Section 4.5.1). This leaves us with a correlation graph containing only the strongly correlated clusters. We then extract all maximal cliques from this graph and transform each clique into a fingerprint. As all maximal cliques are *complete subgraphs*, the edges in the clique do not add any additional information. This

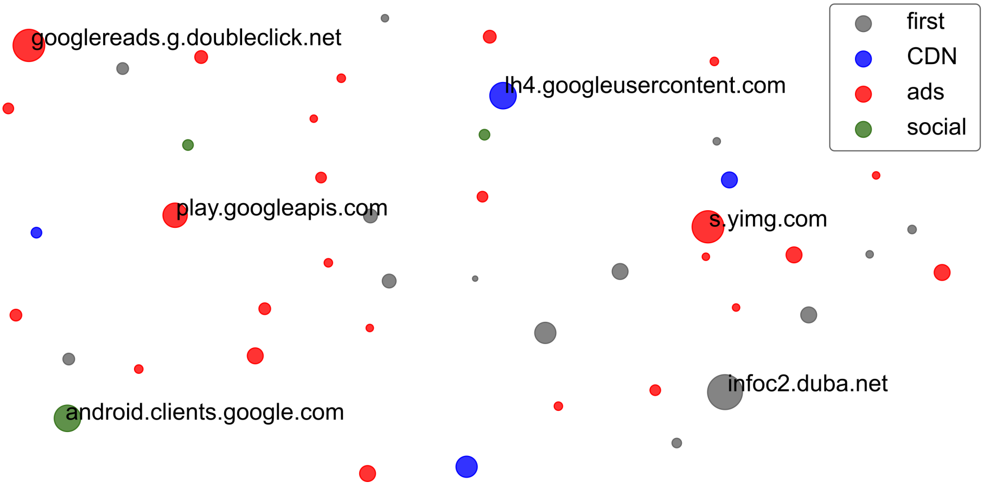


Figure 4.2: **Example of destination clusters for three apps:** *com.rhmsoft.fm*, *com.steam.photoeditor*, and *au.com.penguinapps.android.babyfeeding.client.android*. The size of each cluster is proportionate to the amount of flows assigned to it. We labeled first- and third-party destinations based on the methodology of Ren et al. [177], and distinguished for the latter between CDNs, advertisement networks (ads), and social networks (social).

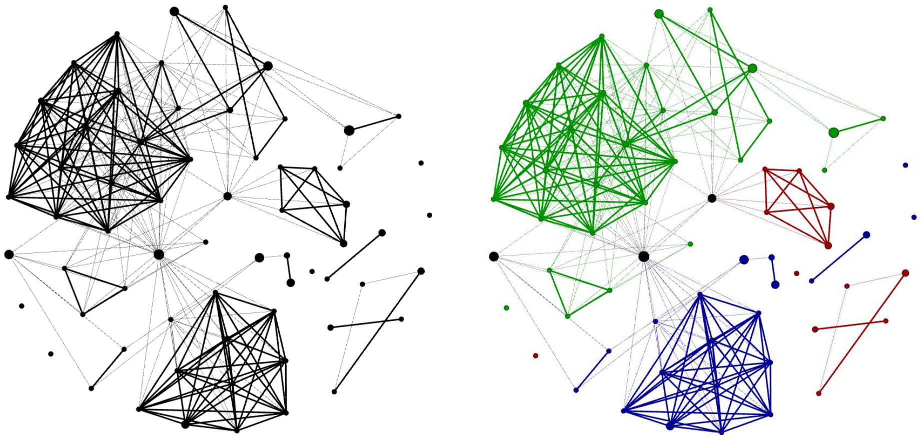


Figure 4.3: **Example correlation graph for three apps as generated by our approach (left) and when labeled per app (right).** The apps include *com.rhmsoft.fm* (blue), *com.steam.photoeditor* (green) and *au.com.penguinapps.android.babyfeeding.client.android* (red) or shared destination clusters (black). Larger nodes indicate the more flows to that destination cluster. The thickness of each edge depends on the cross correlation.

means we can transform cliques into sets of network destinations by extracting all (destination IP, destination port)-tuples and TLS-certificates from every node in a clique and combine them into a set. By performing this transformation for each clique, we obtain all of our fingerprints. In short, we define an app fingerprint as *the set of network destinations that form a maximal clique in the correlation graph.*

As graph edges in the correlation graph depend on the activity of a destination with other clusters, some of the nodes are completely disconnected from the rest of the graph. This is often the case for destinations that are shared among many apps. Figure 4.3 shows an example where the shared (black) nodes only have low cross correlations that fall under the threshold. As these 1-cliques often correspond to multiple apps, treating them as fingerprints yields little added value. However, they will most likely originate from the same app for which we are able to produce fingerprints during the batch processing. Therefore, we assign flows from 1-cliques to the fingerprint that is closest in time, or, if two fingerprints are equally close, to the fingerprint containing the most flows.

### 4.4.6 Fingerprint Comparison

The benefit of using a fingerprint to represent traffic from an app is that it can be computed from the features of the network traffic itself without any prior knowledge. Moreover, we want to compare fingerprints with each other to track app activity over time. Unfortunately, apps communicate with various sets of destinations at different times, either because traffic is based on user interaction, which is dynamic, or because apps produce distinct traffic for different functionalities. Consequently, fingerprints of the same app can diverge to various degrees. To account for this fact, we do not compare fingerprints as an exact match, but instead base their comparison on the Jaccard similarity [108]. Since our fingerprints are sets, the Jaccard similarity is a natural metric to use. To test whether two fingerprints are similar, we compute the Jaccard similarity between two fingerprints $F_a$ and $F_b$ (displayed in Equation 4.5) and check whether it is larger then a threshold $\tau_{\text{similarity}}$. If this is the case, we consider the two fingerprints to be the same.

$$J(F_a, F_b) = \frac{|F_a \cap F_b|}{|F_a \cup F_b|} \tag{4.5}$$

By comparing fingerprints in this way, we are able to track the activity of apps between different input batches and executions of the our approach. In addition, it automatically solves the problem when we observe a fingerprint where one edge of the clique is missing because it did not make the threshold cutoff. Especially when cliques become larger, the possibility of a clique missing an edge increases. In such cases, our approach would output multiple fingerprints for the same app. If these fingerprints are similar, they can even be merged by taking the union of fingerprints.

In addition, this comparison based on the Jaccard similarity allows our approach to treat similar fingerprints as equivalent.

## 4.5 Evaluation

We implemented a prototype of our approach, called FlowPrint, in Python using the Scikit-learn [162] and NetworkX [96] libraries for machine learning and graph computation. The first experiment in our evaluation determines the optimal parameters for our approach. Then, we analyze to what extent the fingerprints generated by our approach can be used to precisely identify apps. Here, we compare our approach against AppScanner [203], a supervised state-of-the-art technique to recognize apps in network traffic. Thereafter, we evaluate how well our approach deals with previously unseen apps, either through updates or newly installed apps. We then detail specific aspects of our approach such as the performance of the browser detector, the confidence level of our fingerprints, and the number of fingerprints produced per app. We further investigate how well our approach can deal with the homogeneous, dynamic, and evolving nature of mobile network traffic. Finally, we discuss the impact of the number of apps installed on the device and demonstrate that our method is able to run in real-time by assessing the execution time of FlowPrint.

**Experimental setup.** Our evaluation requires ground truth labels, which for mobile apps can be acquired by installing an intrusive monitoring agent on a real device, or by running controlled experiments. Due to privacy concerns and to ensure repeatability of our experiments, we evaluate FlowPrint on the datasets described in Section 4.2.1, which closely approach an open world setting, containing encrypted data, user-generated data, both Android and iOS apps, and different app versions. As explained in Section 4.4, our approach does not require any prior knowledge to generate fingerprints, and we leverage ground truth labels only to evaluate our prototype (i.e., to assign app names to matched fingerprints).

We split the traffic of each app in our datasets 50:50 into training and testing sets, without any overlap. For each experiment, we build our database from the training data of 100 randomly chosen apps for each dataset. This leads to an average of 2.0 fingerprints per app for ReCon and Andrubis, and 6.2 for the Cross Platform dataset. For the unseen app detection, we additionally introduce traffic from 20 randomly chosen apps that are not present in the training set.

### 4.5.1 Parameter Selection

As detailed in the previous section, our approach requires four configurable parameters to create fingerprints:

- $\tau_{\text{batch}}$ sets the amount of time of each batch in a network capture to process in each run of our approach.
- $\tau_{\text{window}}$ specifies the time window for destination clusters to be considered active simultaneously.
- $\tau_{\text{correlation}}$ describes the minimum amount of correlation $(c_i \star c_j)_{norm}$ between two destination clusters to have an edge in the correlation graph.
- $\tau_{\text{similarity}}$ indicates the minimum required Jaccard similarity between fingerprints to be treated as equivalent.

**Optimization metric.** We optimize each parameter with respect to the F1-score that our approach achieves when recognizing apps. This metric computes the harmonic mean between precision and recall and is often used to evaluate security solutions. As we output fingerprints, we need to map them to app labels in order to evaluate our approach. Each fingerprint consists of flows which, in our dataset, are labeled. Hence, we label each fingerprint with the flow label that is most commonly assigned to that fingerprint. To illustrate this, suppose fingerprint $F$ contains 10 flows of app $A$ and 2 flows of app $B$, all 12 flows of that fingerprint will be assigned the label $A$. While this approach can generate multiple fingerprints per app (see Section 4.5.4), many security applications (e.g., firewalls) use a mapping on top of fingerprinting and allow multiple fingerprints for the same app.

**Parameter selection.** To optimize our parameters, we refine them individually to reach an optimal F1-score. We choose our parameters from the following set of possible values:

- $\tau_{\text{batch}}$: 1m, 5m, 10m, 30m, 1h, 3h, 6h, and 12h.
- $\tau_{\text{window}}$: 1s, 5s, 10s, 30s, 1m, 5m, 10m, and 30m.
- $\tau_{\text{correlation}}$: 0.1 to 1.0 in steps of 0.1.
- $\tau_{\text{similarity}}$: 0.1 to 1.0 in steps of 0.1.

The batch size thresholds vary between 1 minute, a scenario where apps can be detected while they are still running, and 12 hours, representing a post-incident analysis. The window thresholds vary between 1 second and 30 minutes, where smaller values may miss flow correlations and larger values may correlate flows that were accidentally active around the same time period. Both correlation and similarity thresholds are evenly spread out between 0.1 and 1.0, the first and max values that trigger the corresponding fingerprint mechanism.

Table 4.3: **Summary of tested parameter optimization values.** The first row shows the default parameters and each subsequent row highlights the optimal values found for each individual parameter.

| $\tau_{\text{batch}}$ | $\tau_{\text{window}}$ | $\tau_{\text{correlation}}$ | $\tau_{\text{similarity}}$ | **F1-score** |
|---:|---:|---:|---:|---:|
| 3600 | 5 | 0.3 | 0.5 | 81.64% |
| **300** | 5 | 0.3 | 0.5 | 82.94% |
| 300 | **30** | 0.3 | 0.5 | 83.67% |
| 300 | 30 | **0.1** | 0.5 | 85.43% |
| 300 | 30 | 0.1 | **0.9** | 91.90% |

For each parameter we vary the value by iterating over the test set of possible values while keeping the other parameters as their default value. Once we find an optimal value for a parameter, it is set as the new default for optimizing the other parameters. This way of iterating through the values allows us to capture dependencies between the parameters. To get an average result, we perform a 10-fold cross validation analysis for each setting on held-out validation data from the Andrubis dataset. This held-out data is not used in the remainder of the evaluation to remove bias from this optimization step. We opt to optimize the parameters using only the Andrubis dataset to ensure all datasets contain enough testing data to evaluate our approach. While this may bias the optimal parameters to a specific dataset, our results in the remainder of this section show that the parameters also generalize well to other datasets. During the experiment, we assume that each device has 100 apps installed, which resembles a realistic setting [25]. We also performed the same evaluation with 200 apps per device, which resulted in the same optimal parameters.

As shown in Table 4.3, we find optimal values for $\tau_{\text{batch}} = 300$ seconds, $\tau_{\text{window}} = 30$ seconds, $\tau_{\text{correlation}} = 0.1$ and $\tau_{\text{similarity}} = 0.9$ from this analysis.[2] One interesting observation is that the optimal value for $\tau_{\text{batch}}$ is found at 300 seconds. This means that it may take up to five minutes before a flow is assigned to a fingerprint. In settings that require faster fingerprint generation, operators can of course set a lower $\tau_{\text{batch}}$ value, however at the cost of a lower performance.

### 4.5.2 App Recognition

Many security solutions use a fingerprinting method for the purpose of app recognition [5, 189, 203]. To evaluate the extent to which our approach recognizes apps within network traffic, we create fingerprints of labeled training data. Then, we la-

[2] Due to space limitations we provide additional results about the parameter selection at `https://doi.org/10.4121/e08823b5-ceff-4ebc-a967-290ef9cacc7e`.

Table 4.4: **Performance of our approach compared to AppScanner in the app recognition experiment.** AppScanner was used in Single Large Random Forest mode. The number of flows shown for the Andrubis dataset indicate the minimum number of required flows an app had to produce to be included in the experiment.

| | **Dataset** | **FlowPrint** | | | | **AppScanner** | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **Prec.** | **Recall** | **F1** | **Acc.** | **Prec.** | **Recall** | **F1** | **Acc.** |
| ReCon | normal | 94.70% | 94.47% | 94.58% | 94.47% | 89.60% | 42.84% | 57.97% | 42.84% |
| | extended | 89.84% | 89.22% | 89.53% | 89.22% | 93.65% | 25.34% | 39.89% | 25.34% |
| X-Pltf. | Android | 90.07% | 86.98% | 87.02% | 86.98% | 91.08% | 88.67% | 86.93% | 88.67% |
| | iOS | 94.38% | 92.54% | 92.60% | 92.54% | 85.38% | 14.84% | 24.30% | 14.84% |
| | Average | 91.91% | 89.23% | 89.17% | 89.23% | 87.91% | 50.28% | 57.57% | 50.28% |
| Andrubis | ≥ 1 flow | 58.42% | 58.71% | 58.56% | 58.71% | 62.70% | 19.56% | 29.82% | 19.56% |
| | ≥ 10 flows | 54.39% | 50.31% | 52.27% | 50.31% | 60.69% | 15.01% | 24.07% | 15.01% |
| | ≥ 100 flows | 76.17% | 68.52% | 72.14% | 68.52% | 85.20% | 50.48% | 63.40% | 50.48% |
| | ≥ 500 flows | 73.89% | 74.13% | 74.01% | 74.13% | 86.63% | 53.86% | 66.42% | 53.86% |
| | ≥ 1000 flows | 80.21% | 81.11% | 80.66% | 81.11% | 91.41% | 60.05% | 72.48% | 60.05% |

bel each fingerprint with the app label most commonly assigned to flows within the fingerprint, i.e., we perform a majority vote. After obtaining the labeled fingerprints we run our approach with the test data. We then compare the resulting test fingerprints with the labeled training fingerprints using the Jaccard similarity, as detailed in Section 4.4.6. Subsequently, each test fingerprint, and by inference each flow belonging to that test fingerprint, receives the same label as the training fingerprint that is most similar to it.

We compare our approach with the state-of-the-art tool AppScanner [202, 203]. However, the authors of AppScanner only released precomputed length statistics about the flows in their dataset and the code for running the classification phase on such preprocessed statistics. Therefore, to be able compare both approaches on the same datasets, we faithfully reimplemented the AppScanner feature extraction strategy, which reads PCAP files and feeds the feature values to the classifier.[3] To do so, we followed the description in the AppScanner paper for computing feature statistics, using the standard NumPy [156] and Pandas [139] libraries. AppScanner has different settings, it can either work with a Support Vector Classifier or a Random Forest Classifier. We evaluate AppScanner with a single large Random Forest Classifier, which achieved the highest performance in AppScanner's evaluation. In addition, AppScanner requires a parameter that sets the minimum confidence level for recognition. The optimal confidence level according to the original paper is 0.7,

[3] We release our implementation of AppScanner at `https://doi.org/10.4121/db4fbbb9-fe7d-44b0-b8ec-02a8c81481d9`.

hence this is what we used in our evaluation. Lowering this threshold increases the recall and decreases the precision of AppScanner.

**Comparison with AppScanner.** We evaluate FlowPrint against AppScanner by running a 10-fold cross validation on the same datasets discussed in Section 4.2.1. Additionally, we measure to what extent the performance of our approach is affected by the number of flows produced per app. As apps in the Andrubis dataset produce a varying amount of data, we evaluated the performance considering only apps having a minimum of $x$ flows. This resulted in five evaluations for $x = 1$, i.e., all apps, $x = 10$, $x = 100$, $x = 500$, and $x = 1000$. We refer to these evaluations as *Andrubis* $\geq$ *x flow(s)* for each respective value of $x$. All experiments assumed a maximum of 100 active apps per device in accordance with recent statistics [25].

Table 4.4 shows the performance of both FlowPrint and AppScanner. We note that the accuracy and recall levels are the same, which is due to computing the micro-average metrics for the individual apps. This is often regarded as a more precise metric for computing the precision, recall and F1-score and has the side effect that the accuracy equals the recall [92]. Despite competing with a supervised learning method, we see that both AppScanner and our approach have similar levels of precision, meaning they are able to correctly classify network flows to their corresponding app. However, we outperform AppScanner greatly on the recall, meaning that our approach is much better at classifying all types of traffic, whereas AppScanner provides a sufficient certainty level for only a small fraction of apps. We note that in our experiments, AppScanner has a lower performance than reported in the original paper, especially for the recall. The cause is twofold: First, most apps in our datasets are captured over shorter periods of time, making it more difficult to recognize apps. Second, the AppScanner paper reported only on flows for which they have a confidence level $\geq 0.7$, which in their dataset was 79.4% of flows. This means that unclassified flows are not taken into account. As unrecognized flows reveal much about the recognition approach, our work reports the performance over all flows, where unrecognized flows cause lower recall rates.

**Dataset independence.** Our evaluation shows that FlowPrint performs well on both synthetic (ReCon and Andrubis) and human-generated (Cross Platform) traffic. Furthermore, the results from the Cross Platform dataset show that our approach can be used to generate fingerprints for both iOS and Android apps. However, this does not necessarily mean that a fingerprint generated for an iOS app can be used to detect the corresponding Android version or vice versa. In the Andrubis dataset, we observed no significant difference between recognizing benign and potentially harmful apps. Moreover, the flow experiment (see Table 4.4) shows that apps generating a small amount of flows are more difficult to detect. As a result, our approach has to find correlations between traffic in a limited timeframe resulting in a lower precision. This is a known limitation of network-based approaches and also affects related tools such as AppScanner.

### 4.5.3 Detection of Previously Unseen Apps

In addition to app recognition, we evaluate the capabilities of our fingerprinting approach to detect previously unseen apps. Here, we want FlowPrint to be able to correctly isolate an unseen app as a new app, instead of classifying it as an existing one. This isolation allows also us to distinguish between different unseen apps. Subsequently, when FlowPrint detects a previously unseen app, the security operator can choose to include the new fingerprints in the database. From that point forward, the new app will be classified as known and can be recognized as in Section 4.5.2. For this setting, we create fingerprints for the apps that are present on the device. Subsequently, we add previously unseen apps to the evaluation and generate fingerprints for all the apps present during this testing phase. Our work uses the same parameters from Section 4.5.1 for detecting unseen apps. However, in order to decide whether a fingerprint originates from a new or existing app, we introduce a different threshold $\tau_{\text{new}}$. This threshold indicates the maximum Jaccard similarity between a tested fingerprint and all training fingerprints to be considered a new app. Note that the lower this threshold, the more conservative we are in flagging fingerprints as originating from new apps. The rationale for introducing this additional threshold is that fingerprints remain the same for the entire approach, but are interpreted differently depending on the use case. When detecting unseen apps, we suggest the use of a threshold of 0.1, meaning that only fingerprints that have an overlap of less than 0.1 with all existing fingerprints are considered new apps. Comparing fingerprinting approaches for detecting unseen apps is difficult because, as far as we are aware, the only network-based approaches for detecting unseen apps are DECANTeR [5] and HeadPrint [7]. Unfortunately, both detectors only handle unencrypted data, thus they cannot be applied on encrypted data like ours. Hence, we are unable to compare our approach with related work in this setting.

As in previous experiments, we assume each device has 100 apps installed, and introduce 20 new apps. We evaluate our detector by running a 10-fold cross validation using $\tau_{\text{new}} = 0.1$. A low $\tau_{\text{new}}$ threshold ensures that known apps are not detected as new despite the dynamic nature of apps. As a trade-off, the detector does not correctly classify all flows of previously unseen apps. However, we argue that correctly classifying all flows of unseen apps is infeasible as large parts of many apps are shared in the form of common libraries. This means that it is preferable to aim for a high precision in flows flagged as new apps rather than a high recall as long as previously unseen apps can be detected at some point.

Table 4.5 shows the results of our experiment. We see that the precision is rea-

Table 4.5: **Performance of our approach when detecting unseen apps.** True positives = correctly identified new apps; true negatives = correctly identified known apps; false positives = known apps classified as new; false negatives = new apps classified as known.

| Dataset | Precision | Recall | F1-score | Accuracy |
|---|---|---|---|---|
| ReCon | 97.77% | 70.98% | 82.25% | 85.50% |
| ReCon extended | 99.48% | 20.32% | 33.75% | 54.94% |
| Cross Platform (Android) | 91.06% | 43.18% | 58.58% | 66.34% |
| Cross Platform (iOS) | 96.37% | 77.44% | 85.88% | 85.27% |
| Cross Platform (Average) | 93.52% | 54.49% | 68.86% | 72.53% |
| Andrubis (≥ 1 flow) | 47.57% | 20.90% | 29.04% | 51.00% |
| Andrubis (≥ 10 flows) | 57.03% | 25.52% | 35.26% | 49.65% |
| Andrubis (≥ 100 flows) | 84.05% | 47.60% | 60.78% | 63.86% |
| Andrubis (≥ 500 flows) | 77.22% | 31.21% | 44.46% | 59.15% |
| Andrubis (≥ 1000 flows) | 79.39% | 34.44% | 48.04% | 61.77% |

sonably high and 97.8% of flows are correctly flagged as unseen for ReCon and 99.5% for ReCon extended. This also means that existing apps are rarely marked as unseen, reducing the load on any manual checking of alert messages. On the Cross Platform dataset, we achieve 93.5% precision on average indicating that, while slightly more difficult, our approach is still capable of detecting new apps without raising too many false alerts. For Andrubis, the rate of false positives is higher with 14.8% for apps producing at least 100 flows. This is due to the relatively short time span in which traffic of this dataset was produced, i.e., 240 seconds.

**Recall.** We see that the recall is significantly lower than the precision, only reaching 20.3% for the ReCon extended dataset. This is caused by homogeneous behavior of mobile apps, i.e., the network traffic of these apps overlaps due to the use of common libraries and services. In the experiments of Table 4.5 we found that unknown apps showed similar advertisement traffic to known apps. When the similarity ot the unknown app results in a higher matching score than $\tau_{\mathsf{new}}$, it will be misclassified as known. This is less of a problem in the app recognition scenario where FlowPrint searches for a best match. Multiple training fingerprints can have a similarity score > $\tau_{\mathsf{new}}$, but the actual app likely produces the highest score due to most overlapping destinations, leading to a correct match. We elaborate on the effects of homogeneous traffic in Section 4.5.5. As stated before, low recall is not necessarily problematic as long as the unseen app is detected at some point. In our experiment, we already detect 72.3% of apps in the first batch (five minutes) in which they appear. We discuss further limitations of app fingerprinting in Section 4.6.

### 4.5.4 Fingerprinting Insights

In the previous experiments we demonstrated that our approach works for both recognizing already seen apps, as well as detecting unseen apps. In this section, we evaluate specific parts of our fingerprinting approach to give insights into possible other use cases.

**Browser isolation.** We first highlight the performance of the *browser detector* component within our approach. In this experiment we use both the browser dataset and the Andrubis dataset as discussed in Section 4.2.1. As the browser detector is supervised, it performs better when trained with a large set of applications, hence the Andrubis dataset is a natural choice for this evaluation. To this end, we randomly selected 5,000 non-browser apps from the Andrubis dataset to represent non-browser data. Of these apps, we used an 80:20 split for training and testing our detector respectively. Recall that when we detect a browser, all flows within a surrounding 20 second window are marked as browser traffic. This window was empirically optimized to achieve high recall rates. To ensure that wrong detections are properly penalized in our experiment, we interleave the browser and non-browser traffic by shifting all timestamps such that each trace starts at the same time.

Note that, while there exist apps that embed a "browser window" (e.g., Android `WebView`), we do not consider these apps as browsers because of their confined access to a limited set of network destinations. In contrast, real browsers navigate to many different websites, producing a bigger relative change of active clusters—one of the features of our browser isolation. In fact, our datasets contain several HTML5 apps, which we correctly detected as regular apps.

Table 4.6 shows the average performance of the browser detector using ten Monte Carlo cross validations. Our detector achieves, on average, an accuracy of 98.1% and detects browser flows with a recall of 98.3%. Unfortunately, with a precision of 79.8% the number of wrongly isolated streams is rather high due to the aggressive detection. This in turn leads to 1.8K of 25.8K non-browser clusters being incorrectly removed at some point. Fortunately, 75.7% of these clusters resurfaced after the initial removal without being mistakenly detected as a browser. This means they are still used for fingerprinting their corresponding non-browser apps. In total only 1.7% of non-browser clusters were permanently removed.

**Confidence.** FlowPrint assigns unlabeled fingerprints to each passing flow. To gain more insights into how these fingerprints are represented we assign a confidence level to each fingerprint that measures how certain we are that each flow within a fingerprint belongs to the same app. In order to measure confidence, we

Table 4.6: **Performance of the Browser Detector based on the number of detected TCP/UDP streams.**

| | **Actual Browser** | **Actual non-Browser** |
|---|---:|---:|
| **Predicted Browser** | 21,987 (TP) | 5,574 (FP) |
| **Predicted non-Browser** | 363 (FN) | 28,4125 (TN) |

Table 4.7: **Confidence levels of our fingerprints.** A score of 1 indicates fingerprints only contain flows of a single app.

| **Dataset** | **Confidence** |
|---|---:|
| ReCon | 0.9857 |
| ReCon extended | 0.9670 |
| Cross Platform (Android) | 0.9740 |
| Cross Platform (iOS) | 0.9887 |
| Cross Platform (Total) | 0.9864 |
| Andrubis | 0.9939 |

look at the amount of information gained by knowing to which fingerprint a flow belongs to with respect to the app label of that flow. That is, we measure by what fraction the entropy of app labels is reduced if we know the fingerprint of each flow. Equation 4.6 shows the formula for computing the confidence of our fingerprints. Here, $H(A|F)$ is the entropy of app labels for each flow, given that we know its fingerprint. $H(A)$ is the entropy of the labels without knowing the fingerprints. When all fingerprints only consist of flows of a single app knowing that fingerprint automatically leads to knowing the label. Therefore, $H(A|F) = 0$ gives a confidence level of 1. In case knowing the fingerprint does not provide additional information regarding the app label of a flow $H(A|F) = H(A)$ and therefore, the confidence level is 0. In clustering, this is referred to as homogeneity [180].

$$\text{Confidence} = 1 - \frac{H(A|F)}{H(A)} \tag{4.6}$$

Table 4.7 shows the confidence level of fingerprints produced by our approach for each dataset. We see that for each dataset we achieve confidence levels close to 1 meaning that the majority of our fingerprints contain only flows of a single app.

**Cardinality.** Each app is ideally represented by a single fingerprint. This would make it possible to automatically separate the network traffic into bins of different apps. However, this might be infeasible as mobile apps offer different functionalities which may result in multiple fingerprints. Therefore, we also investigate the number of fingerprints our approach generates for each app. We recall that an app can be viewed as a combination of individual modules, including third-party libraries, that each account for part of the app's functionality. This naturally leads to apps presenting multiple fingerprints. We refer to the number of fingerprints generated per app as the *cardinality* of each app.

Figure 4.4 displays the cardinality of apps in our datasets and shows that the majority of apps in all datasets have multiple fingerprints. Our previous evaluations have shown that this is not a problem for the app recognition and unseen app detection settings. However, the cardinality of apps in our work should be taken into account in case a new app is detected. Here, security operators should be aware that there will likely emerge multiple fingerprints for that new app. We note that the ReCon extended dataset is not shown in this graph since all apps in that dataset had more than 20 fingerprints. This is in large part due to the fact that apps in the ReCon extended dataset contain more versions, which introduce additional fingerprints (also see Section 4.5.5). On average, each version in the ReCon extended dataset contained 18 fingerprints. This number of fingerprints per version is still higher than the other datasets because each app was exercised longer, leading to more app functionality being tested, which in turn led to more fingerprints. Finally, Figure 4.4 shows that apps in the Cross Platform dataset have a higher average cardinality than the other datasets. This suggests that user interaction leads to more fingerprints describing individual app functionalities rather than the entire app itself.

### 4.5.5 Mobile Network Traffic Challenges

We evaluate the effect of the three properties (Section 4.1) of the mobile network traffic that pose challenges for our approach: its *homogeneous*, *dynamic* and *evolving* nature.

**(1) Homogeneous traffic.** The first challenge is that mobile traffic is homogeneous because traffic is encrypted and many apps share the same network destinations, for example due to shared third-party libraries, or the use of CDNs and common cloud providers. In this experiment, we analyze to what extent the homogeneity caused by shared network destinations affects the performance of our approach. We analyzed the ReCon dataset, which includes DNS information for each flow, as well as a classification of each DNS address as a first-party or third-party destination for each app, allowing us to investigate the cause of homogeneity. In detail, this classification maps domains, and by extension flows, to one of the following categories based on properties of the app's description in the Google Play Store and WHOIS

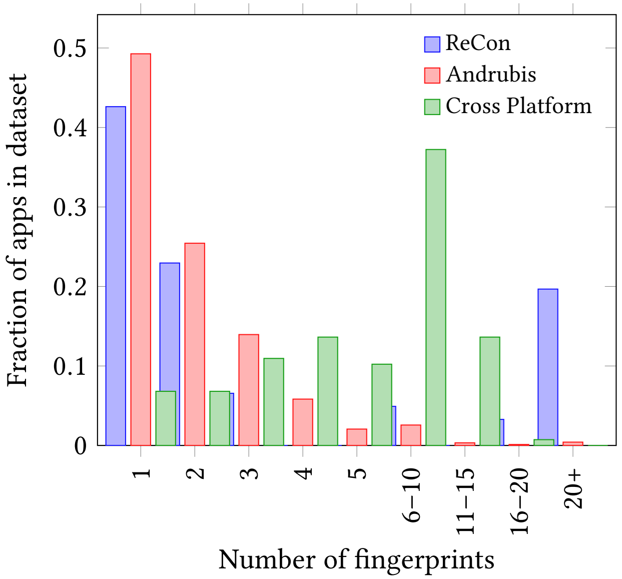


Figure 4.4: **Number of fingerprints (cardinality) generated per app.**

information [177]: (1) *first-party*, i.e., app-specific traffic, and *third-party* traffic. For the latter we further distinguish between (2) CDN traffic, (3) advertisement traffic, and (4) social network traffic, based on publicly available adblocker lists, extended by manual labeling. In turn, we classify each cluster according to a majority vote of the flows within that cluster.

Our experiment found a total of 2,028 distinct clusters, of which 281 clusters are shared between more than one app. At first sight, the homogeneity of traffic seems quite low with only 13.9% of all clusters being shared. However, these shared clusters account for 56.9% of all flows in the dataset. By looking at the categories, we find that advertisement networks account for 60.6% of traffic spread over 184 different shared destination clusters. As apps often use standard libraries for displaying advertisement it is unsurprising that many flows are homogeneous with respect to their network destination. Social networks account for 30.4% of traffic to shared clusters. Similar to advertisements, the support for social services is often provided by commonly used libraries such as the Facebook SDK[4] or Firebase SDK[5]. Finally, we find that 6.0% and 2.9% of shared cluster traffic originates from app-specific network destinations and CDNs respectively.

Then, we evaluate how our approach reacts under higher levels of homogeneity. To this end, we removed all flows that are not shared between apps from the

[4] https://developers.facebook.com/docs/android/

[5] https://firebase.google.com/docs/auth/android/start

ReCon dataset, leaving *only* shared clusters. When running our approach for recognizing apps, the F1-score drops from 94.6% to 93.0% and accuracy drops from 94.5% to 93.3%. Despite the small drop in performance, we are still able to accurately distinguish apps because the different correlation patterns of these shared clusters can still be uniquely identified. Therefore, our approach shows robustness against homogeneous network traffic.

**(2) Dynamic traffic.** The second challenge is the dynamic nature of the traffic generated by users as they interact with apps by using different functionalities at different times. In contrast, automatically generated datasets often aim to cover as much functionality as possible in a short amount of time. This difference between datasets may lead to a different quality of the fingerprints. To evaluate whether our approach is influenced by dynamic traffic, we look at the performance difference of our approach between the user-generated Cross Platform dataset and the other datasets. Although these datasets are not directly comparable due to the different apps they contain, we do not find a significant difference in the detection capabilities of our approach (see Tables 4.4 and 4.5). We attribute this in part to the amount of network traffic produced by apps without requiring any user interaction. These include connections to, for example, advertisement and social networks, as well as loading content when launching an app. The high performance for both recognizing apps and detecting unseen apps from user-generated traffic suggests that dynamic traffic does not impose any restrictions on our approach.

**(3) Evolving traffic.** The final challenge concerns the evolving nature of apps. Besides detecting previously unseen apps (Section 4.5.3), we evaluate our approach when dealing with *new versions of an existing app*, and we perform a longitudinal analysis to assess how FlowPrint performs when the values of our features change over time.

**(3a) App updates.** We use the ReCon and ReCon extended datasets as they contain apps of different versions released over 8 years. On average, the datasets contain 18 different versions per app, where new versions were released once every 47.8 days on average. As the traffic of these different versions was captured over a period of two and a half months, changes in IP addresses and certificates might cause a slight bias in the dataset. In the next subsection, we describe the results of our longitudinal analysis, which provides a more in depth analysis regarding this influence. Nevertheless, we demonstrate that new app functionality introduced by updates does not necessarily cause an issue with our fingerprints if caught early. For this experiment, we train the unseen app detector with a specific version of the app as described in Section 4.5.3. In turn, for each newer version of the app, we

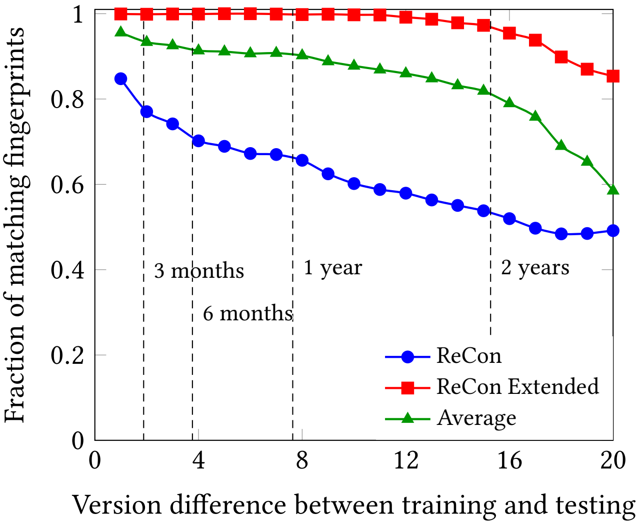


Figure 4.5: **Recognition performance of FlowPrint between versions.** The x-axis shows the number of different versions, including the average time apps take to receive so many version updates. The y-axis shows the fraction of matching fingerprints between training and testing data.

run the unseen app detector to predict whether the fingerprints of this new version match the training version. We perform this experiment by increasing the amount of versions between the training data and the version to predict. This simulates a security operator lagging behind in updating the models and thus missing intermediate versions.

Figure 4.5 shows the results of this experiment. Here, the x-axis shows the amount of versions between the training app and predicted app. The y-axis shows the relative number of fingerprints from the newer app versions that FlowPrint correctly recognizes. As we know the average amount of time it takes for an app to be updated (47.8 days), we show the decline in performance not only in terms of versions, but also over time, by the vertical dashed lines in the plot. We found that on average FlowPrint recognizes 95.6% of the fingerprints if FlowPrint is updated immediately when a subsequent version is released. When we do not update immediately, but wait a certain amount of time, the detection rate slowly drops, which is more evident in the ReCon dataset. The difference between the two datasets is due to (1) more traffic per app in the ReCon Extended dataset, which makes fingerprinting more accurate, and (2) a larger set of apps in the ReCon dataset, which makes recognition more difficult. The average result shows the analysis for the combined datasets and gives the most realistic performance, which shows that FlowPrint can recognize 90.2% of the new fingerprints even when operators do not update the models for one year. Interestingly, 45.5% of the apps in our datasets released multiple new versions on the same day. However, FlowPrint showed nearly identical performance for these same-day updates, leading us to believe that quick version releases do not introduce major app fingerprint changes.

**(3b) Longitudinal analysis.** Over time, the destination features (IP address, port) and the TLS certificate may change because of server replication/migration or certificate renewals. To measure how FlowPrint's performance changes over extended periods of time, we evaluate how feature changes affect our approach. To do this, we train FlowPrint using the original training data and consistently change a percentage of random IP addresses and TLS certificates in the testing data. As TLS certificates are domain-based and not IP-based, random selection gives a good approximation of FlowPrint's performance. We performed a 10-fold cross validation changing 0 to 100% of such features in steps of 10% points.

Figures 4.6 and 4.7 show the performance of FlowPrint, in the case of app recognition and unseen app detection respectively, for an increasing amount of changes in our features. As with the app updates, we indicate the expected amount of changed features after given periods of time by the vertical dashed lines. These expected changes are computed from the average lifetime of certificates in our dataset and DNS-name-to-IP changes according to the Farsight DNSDB database [188]. For the case of app recognition, the number of changed features initially has limited effect because changing only one of the two destination features still allows our clustering approach to detect the same network destination. Once we change approximately 80% of the features, the decline becomes a lot steeper because at this point both features are changed simultaneously. When changing 100% of IP addresses and certificates we are unable to detect anything. Interestingly, the Andrubis performance of the dataset declines almost linearly. That is because only 24.7% of Andrubis flows contain a TLS certificate. Hence, the certificate cannot counteract changes in the IP address, leading to a steeper decline. This also underlines the importance of using both the IP and TLS certificate as destination features. We recall from Section 4.4.2 that destination features may be enriched by domains from DNS traffic. As domains are generally more stable than IP addresses, they will have a positive effect on the performance over time. For the case of unseen app detection, an increase in changed features leads to an increase in the recall. After all, if traffic of a previously unseen app differs more from the training data, the app will be flagged as previously unseen. For the same reason, the detection precision declines as known apps increasingly differ from their training dataset.

Subsequently, we performed a real-world experiment by collecting and analyzing data from the current versions of 31 apps in the Cross Platform dataset more

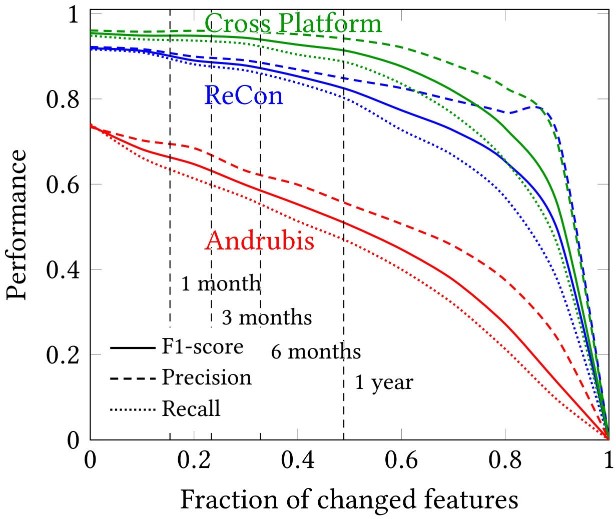


Figure 4.6: **App recognition performance vs changes in both IP and certificate features.** The x-axis denotes the % of changed features. Where the expected amount of change over time is denoted by the dashed vertical lines.

than 2 years (26 months) after the original capture. When FlowPrint trains on the original dataset and performs recognition on the recollected flows it achieved a precision of 36.7%, recall of 33.6% and F1-score of 35.1%. This translated to being able to recognize 12 out of 31 apps. Interestingly, if we only look at the apps that we were able to recognize, FlowPrint performs with a precision of 76.1%, a recall of 62.2% and an F1-score of 68.4%. The expected decline in performance after 2+ years that we found in our two previous analyses is in line with the results from this real-world experiment.

In conclusion, while, as expected, FlowPrint's performance degrades when a large amount of destination-based features change (i.e., after one year), our approach can cope with a significant amount of variations without drastic performance degradations. We believe this gives operators enough time to update FlowPrint's models to maintain high performance, making our approach practical.

### 4.5.6 Training Size

So far we assumed each device in the network to have 100 apps installed, however FlowPrint may perform better or worse in case this number differs. To evaluate the effect of the number of installed apps, we train FlowPrint by varying the number of apps $N$ in the training data. Recall that our approach builds its model on a per-device basis. Hence, while our database may contain millions of fingerprints, FlowPrint

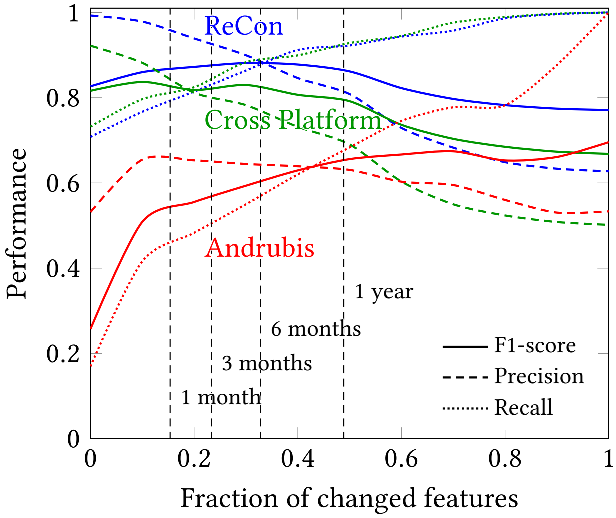


Figure 4.7: **Unseen app detection performance vs changes in both features.** The x-axis denotes the % of changed features. Where the expected amount of change over time is denoted by the dashed vertical lines.

*only matches fingerprints against apps installed on the monitored device.* We train FlowPrint with $N$ apps, ranging from 1 to 200 for the ReCon and Cross Platform datasets, which is already much higher than the average number of installed apps on a device [25]. For the Andrubis dataset, we range $N$ from 1 to 1,000 to evaluate the extreme scenario. We first analyze the performance of our approach in app recognition on the testing data of the same apps. In the second experiment, for each $N$ we introduce 20% previously unseen apps, which FlowPrint has to correctly detect. All experiments use 10-fold cross validation.

Figure 4.8 shows the performance of the different datasets in app recognition. Here we see that for all datasets, the performance of all metrics initially decreases, but stabilizes after a certain point (note that the y-axis starts from 0.85). Even up to the tested scenario of 1,000 apps, for the Andrubis dataset, the F-1 score remains constant at 0.9. This indicates that FlowPrint easily discerns between relatively few apps, because it can still rely on network destinations to differentiate between apps. However, once apps start to share network destinations, the performance drops slightly and quickly stabilizes. Once stabilized, FlowPrint leverages temporal correlations in network destinations found by our correlation-graph, which provide a much more robust way of recognizing apps. We see the same mechanism, although to a lesser degree, for the unseen app detection scenario in Figure 4.9. Here the recall is initially affected because FlowPrint only detects an app as previously unseen if its fingerprint differs enough from the existing ones. When the train-

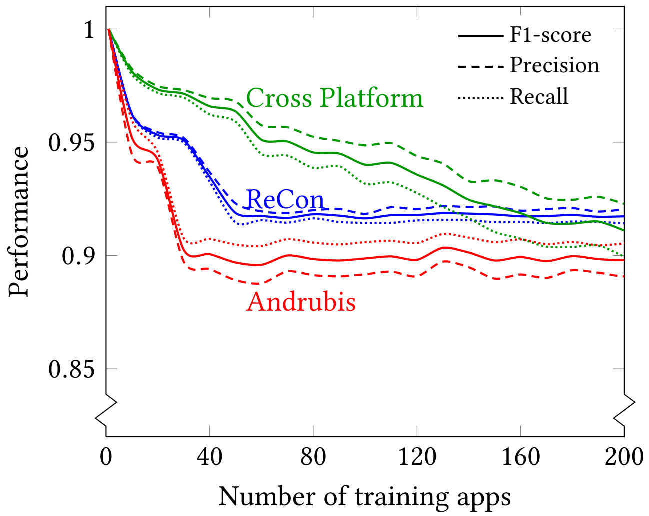


Figure 4.8: **App recognition performance vs training size.**

ing data includes more shared destinations, the probability that a new app overlaps with the original dataset becomes larger, and therefore the detection rate, initially, slightly decreases. Once the training data contains a sufficient amount of shared destinations the performance becomes more consistent. The fluctuations are due to apps producing traffic to shared clusters, which occasionally produce incorrect matches with known apps. Finally, we note that the Andrubis dataset performs notably worse than the other datasets because it contains apps that produce relatively few flows. This is in accordance with the results found in Table 4.5.

### 4.5.7 Assessment of Execution Time

In addition to the aforementioned metrics, the effectiveness of our approach in a real environment also depends on its execution time. As we employ some seemingly high-cost operations, such as clustering and clique discovery, we also assess the individual components of our approach to better understand the actual time complexity involved. We note that, due to the setup of our approach, its complexity depends on the number of network flows rather than the amount of communicated bytes. In order for our approach to run smoothly, it should be able to process all received flows within each batch time $\tau_{\text{batch}}$, which in our prototype is set to five minutes. We assessed the execution time of FlowPrint by running it on a single core of an HP Elitebook laptop containing an Intel Core i5-5200U CPU 2.20GHz processor.

Figure 4.10 shows the average performance over 10 runs of FlowPrint when

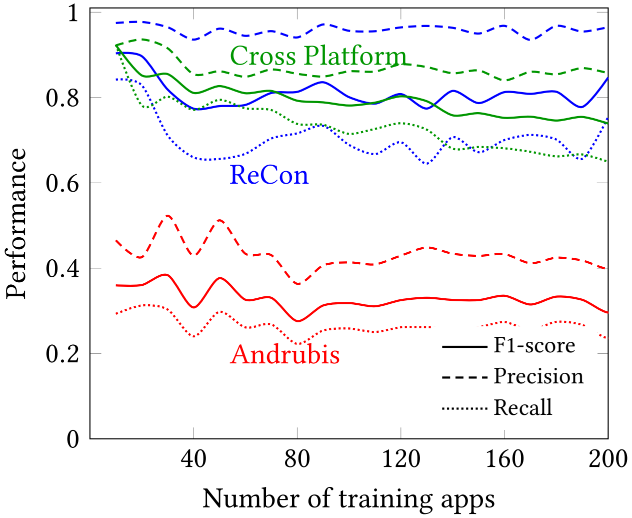


Figure 4.9: **Unseen app detection performance vs training size.**

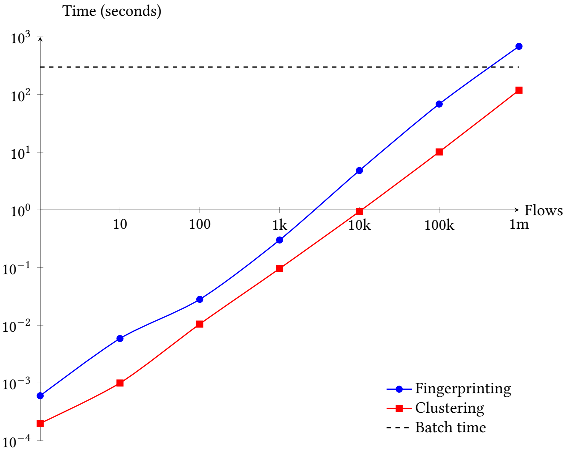


Figure 4.10: **Average execution time of FlowPrint when fingerprinting** $n$ **flows in a single batch.** The fingerprint generation time includes clustering.

generating fingerprints. Here we find that our prototype is able to process roughly 400k flows within the time window of five minutes. To put this number into perspective, the ReCon and Andrubis datasets contain an average of 117 and 22 flows and a maximum of 845 and 1,810 flows per five-minute interval respectively. This means that at peak communication activity FlowPrint is able to handle 221 devices simultaneously on a mid-range laptop, making our approach feasible to run in practice. Both the clustering and cross-correlation have a theoretical time complexity of $O(n^2)$, however, from Figure 4.10 we see that in our approach these components act almost linearly. For the clustering, each flow is clustered together with flows containing the same destination (IP, port)-tuple or the same TLS certificate. Our prototype implements these checks using a hashmap giving the clustering a linear time complexity. For the cross-correlation we note that flows that have the same activity pattern $c[0]...c[T]$ have a mutual cross-correlation of 1 and the same correlation with respect to other flows. Hence, they only need to be computed once, reducing the time complexity.

Generated fingerprints need to be matched against a database of known fingerprints. We consider two scenarios: (1) finding the closest matching fingerprint (for app recognition), and (2) checking for any match (in case of unseen app detection). Figure 4.11 shows the average performance over 10 runs for matching 1,000 generated fingerprints against a database of size $n$. The complexity of matching fingerprints grows both with the database size and the amount of fingerprints matched against this database. Figure 4.11 shows that even for databases containing one million fingerprints, the required time to match is 73 seconds, which is well beneath the five-minute mark of $\tau_{\text{batch}}$. Assuming an average of 100 apps per device and a high cardinality of 20 fingerprints per app (see Section 4.5.4), a database containing one million fingerprints would be able to deal with 500 devices simultaneously on a mid-range laptop. These results suggest the feasibility of our approach in high-volume traffic scenarios as well.

## 4.6 Discussion

We have shown that our approach succeeds in creating semi-supervised fingerprints for mobile apps, and that such fingerprints can be used for both app recognition and detecting previously unseen apps. Nevertheless, there are some aspects of our approach that should be addressed in future work.

**Potential for evasion.** We construct our fingerprints based on the set of network destinations, and the timing of communication with such destinations. In order for authors of an adversarial app to evade detection by our approach, they have two options. First, they may redirect all traffic of their app using a VPN or proxy.

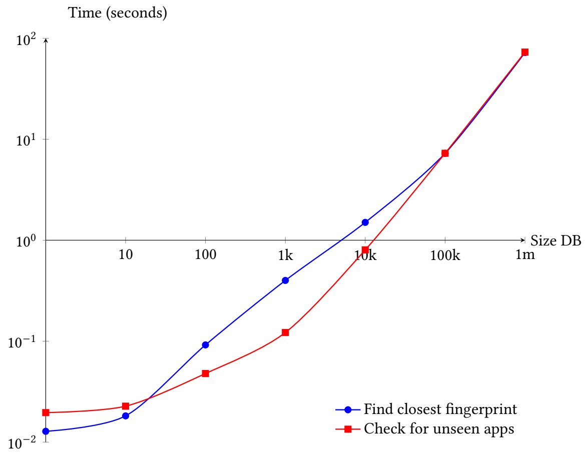


Figure 4.11: **Average execution time of FlowPrint when matching 1,000 fingerprints against a database containing *n* fingerprints.**

When doing this only for their app and not system-wide, its single destination would still show up as a fingerprint, thus that specific app can still be detected. Setting a system-wide proxy or VPN connection for all apps on the device (1) requires manual confirmation by the user; and (2) would be recognizable as unusual device behavior as our approach would detect all device traffic as originating from a single app. Hence, with this evasion technique our approach would still be able to detect the presence of an unknown app but it will have trouble identifying the specific app. The second option is to either avoid producing network traffic (limiting the damage of potentially harmful apps), or to try to simulate the traffic patterns of a genuine app. We expect that being restricted to use the same set of destinations and timing of an existing genuine app severely limits the potential for an attack, especially if the attacker does not have control over such destinations.

**Low-traffic apps.** During our evaluation, we observed cases of apps that cannot be reliably fingerprinted using our approach. This includes, in particular, apps that only communicate with widely used services, e.g. advertisement networks and CDNs, which may be difficult to fingerprint. After all, our fingerprints rely on patterns shown in network destinations. If the pattern generated by an app is common to many other apps, we cannot discern said specific app. We mainly observed this behavior in apps that do not require any form of server for their main functionality, but that still communicate with advertisement and analytics services, probably as a way for monetization. Unfortunately, we expect most network-based monitoring approaches to suffer from the same limitation due to the generic nature of advertisement and analytics communication.

**Simultaneously active apps.** A limitation of a semi-supervised approach is that it has difficulty distinguishing multiple apps that are running at the same time. Android allows apps to exchange network traffic in the background, although this behavior is typically found only in a limited set of apps (i.e., music streaming apps, and apps to make phone calls). In addition, since Android 7, two apps can be in the foreground at the time by splitting the screen of the device. Furthermore, Android 10 allows those apps also to be active simultaneously [190]. We expect this heavy multi-app scenario to create challenges for our fingerprinting approach, and therefore, future work needs to investigate the fingerprint generation for multiple simultaneously active apps.

**Repackaged apps.** While one of our datasets, the Andrubis dataset, also contains flows from potentially harmful and malicious apps, we did not specifically investigate the effect of repackaged apps on our fingerprinting. As malware authors frequently repackage benign apps with their malicious payload [127], it would be interesting for future work to investigate whether the additional fingerprints introduced by this payload could be used to detect this type of malware.

**Fingerprint coverage.** Our evaluation has shown an app may have multiple fingerprints. When detecting new apps, it takes some time for our approach to converge to a state where a sufficient number of fingerprints has been created to accurately characterize the network traffic of an app. Continella et al. [62] already observed this as a limitation when dealing with unknown traffic. Future work could explore approaches similar to theirs to automatically decide when enough network traffic has been fingerprinted to sufficiently cover the network behavior of an app. Furthermore, while the fingerprints of previously unseen apps can be immediately used to recognize the same apps later on, if an unseen app produces multiple fingerprints, FlowPrint recognizes each fingerprint as a separate app. Future work could explore approaches to automatically determine whether a burst of new fingerprints belong to the same previously unseen app.

**AppScanner reimplementation.** While we faithfully reimplemented AppScanner following the approach described in the original paper, our implementation might still slightly differ from the original tool. Therefore, it is possible that the two implementations have slightly different performances. However, we expect this difference to be minimal, if present.

**Privacy implications.** One of the advantages of our work is that it works on encrypted traffic. One can argue that in enterprise networks, TLS can be decrypted by deploying man-in-the-middle TLS proxies and therefore other approaches are still applicable. However, traffic decryption weakens the overall security [77] and violates users' privacy, thus we believe it should be avoided. At the same time, our approach shows the high precision with which apps can be identified despite traffic encryption. From a privacy perspective, the use of certain apps can reveal information about medical conditions, religion, sexual orientation, or attitude towards the government of users. Identifying individual apps from the network traffic alone also opens the door for censorship and traffic differentiation [125]. Furthermore, individuals may be identified and tracked to a certain degree based on the unique set of apps they are using [14]. Since devices from different vendors and carriers often introduce a unique set of pre-installed apps [87], it should at least be feasible to identify a specific device manufacturer or type, which we leave for future work.

## 4.7 Related Work

Related work already explored the use of network fingerprints for both mobile and desktop devices. However, related approaches are either supervised, i.e., require prior training on labeled apps, or only work on unencrypted network traffic.

**App recognition.** App recognition, also referred to as traffic classification, is closely related to app fingerprinting as both approaches attempt to map traffic to the app that produced it. Related work suggested the use of deep packet inspection (DPI) for this purpose. Some approaches attempt to automatically identify clear-text snippets in network traffic that are unique to an app [208, 221]. Other classifiers focus specifically on HTTP headers in combination with traditional machine learning [145] or deep learning approaches [53]. Choi et al. [55] even suggested automatically learning the optimal classifier for each app. As app recognition can only be used for apps for which a fingerprint exists, several approaches extended HTTP-based fingerprints by automating the process of fingerprint creation [52, 65]. However, all these approaches rely on DPI, meaning that they cannot be used on encrypted traffic. Given that 80%–90% of Android apps nowadays communicate over HTTPS, i.e., use TLS [91, 174], any fingerprinting solution should be able to deal with TLS-encrypted traffic.

AppScanner [203] uses statistical features of packet sizes in TCP streams to train Support Vector and Random Forest Classifiers for recognizing known apps. This system is able to re-identify the top 110 most popular apps in the Google Play Store apps 99% accuracy. However, to achieve these results, AppScanner only makes a prediction on traffic for which its confidence is high enough. This results in the system only being able to classify 72% of all TCP streams. BIND [149], like AppScanner, creates supervised app fingerprints based on statistical features of TCP streams. BIND also uses temporal features to better capture app behavior and reaches an average accuracy of 92.6%. However, the authors observed a decay in performance over time, and suggest to retrain the system periodically if lower performance is observed.

Concurrent to our work, Petagna et al. [165] demonstrated that individual apps can also be recognized in traffic that is anonymized through Tor. Their supervised approach uses timing, size, packet direction and burst features of TCP flows. Similar to our work, the authors observed web browsers posing a particular challenge, since each visited website might produce different patterns.

Other approaches include the use of Naïve Bayes classifiers in combination with incoming and outgoing byte distributions [131], the use of statistical flow features in combination with decision trees [34] and the possibility of combining existing classifiers [13]. Alan et al. [16] train a classifier on the packet sizes of the launch-time traffic of apps. However, as the authors acknowledge, detecting the launch of an app in real-world traffic is challenging, and and app might already be launched when a phone enters a network.

Finally, several techniques attempt to identify not the apps themselves, but rather user activity within apps [61, 183]. These methods are able to detect even more subtle differences within app usage which can subsequently be linked to the original app. Unfortunately, none of these approaches address the inherent flaw of app recognition, namely the inability to recognize previously unseen apps.

**Real-time fingerprint generation.** Related approaches on real-time fingerprint generation for the detection of apps either require decrypted network traffic, or focus on detecting the application-layer instead of the mobile app itself. Bernaille et al. [40] stressed the importance of fast recognition of apps in network traffic and suggested the clustering of TCP flows based on their first five messages. Their approach recognizes the application-layer protocol, which might be sufficient for detecting desktop apps. In contrast, since mobile apps mostly communicate over either HTTPS or QUIC, this method is insufficient in our setting. DECANTeR [5] builds desktop app fingerprints from the headers of HTTP messages without requiring prior knowledge of apps. However, this approach also relies on decrypted traffic for fingerprint generation.

**TLS fingerprinting.** In addition to app fingerprinting, TLS fingerprinting techniques are often used to track communicating processes [17, 23, 137, 138]. These techniques leverage the diversity of fields in `ClientHello` messages generated by different TLS implementations to create fingerprints. However, they do not work well in the homogeneous mobile setting where many apps use the same SSL/TLS implementation provided by the underlying OS. Consequently, different apps produce the same TLS fingerprints making it impractical to recognize apps or discover previously unseen apps. This property is even exploited by tools [86] to bypass censorship systems. TLS fingerprinting may also be applied on the `ServerHello` message as done by JA3S [17]. In this setting, it is not the app that is fingerprinted but rather the destination communicating with the app. This technique can potentially be used to improve our destination clustering step, but is not directly applicable to fingerprinting mobile apps. In general, destination-based TLS fingerprinting techniques that focus on desktop applications do not work well when directly applied to mobile apps because, as shown in Section 4.5.5, mobile apps often share destination-based clusters (e.g., advertisement networks).

**Malware detection.** We showed the value of our approach in the setting of unseen app detection, where we treat new apps as potentially malicious. This decision can be made by complementing techniques that focus specifically on classifying malicious traffic [22, 24]. These approaches are not capable of discriminating between individual apps, but rather make a decision on whether traffic contains malicious patterns. Hence, our approach complements these techniques by providing more insights into the individual apps active on the network.

## 4.8 Conclusion

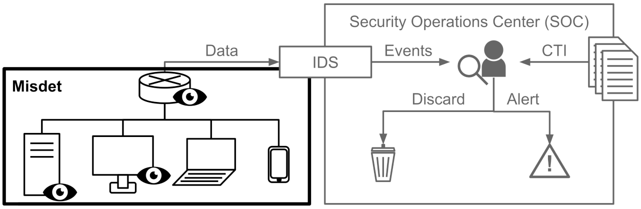
In this chapter we proposed FlowPrint, a novel approach for creating real-time app fingerprints from the encrypted network traffic of mobile devices. Unlike existing approaches, our approach does not rely on any prior knowledge about apps that are active in a certain network. Therefore, the strength of FlowPrint lies in its ability to detect previously unseen apps in encrypted network traffic. This allows us to deal with evolving sets of apps, opening many security applications for which fingerprinting was previously unsuitable.

In our evaluation, FlowPrint achieved an accuracy of 89.2% for recognizing apps, outperforming the supervised state-of-the-art approach. Furthermore, we showed that our approach is able to detect previously unseen apps with a precision of 93.5%. These results demonstrate the capabilities of semi-supervised approaches when dealing with evolving systems, such as mobile apps, even in the presence of largely homogeneous traffic due to third-party libraries and services.

# Chapter 5

## Detecting Anomalous Misconfigurations in AWS Identity and Access Management Policies

We have shown that our context-based approach to the identification and explanation of security events can assist security operators in their daily activities. However, so far, our approach has been reactive, meaning that security incidents can only be caught after they occur. Instead, we wanted to study how contextual analysis could be applied as a preventive measure. Specifically, we performed the following case study:



**Case Study.** *To what extent can context-based analysis assist in preventing misconfigurations in IAM Policies?*

In recent years, misconfigurations of cloud services have led to major security incidents and large-scale data breaches. Due to the dynamic and complex nature of cloud environments, misconfigured (e.g., overly permissive) access policies can be easily introduced and often go undetected for a long period of time. Therefore, it is critical to identify any potential misconfigurations before they can be abused. In this chapter, we present a novel misconfiguration detection approach for identity and access management policies in AWS. We base our approach on the observation that policies can be modeled as permissions between entities and objects in the form of a graph. Our key idea is that misconfigurations can be effectively detected as anomalies in such a graph representation. We evaluate our approach on real-world identity and access management policy data from three enterprise cloud environments. We investigate the effectiveness of our approach to detect misconfigurations, showing that it has a slightly lower precision compared to rule-based systems, but it is able to correctly detect between 3.7 and 6.4 times as many misconfigurations.

This chapter is based on the publication: [2] Thijs van Ede et al. "Detecting Anomalous Misconfigurations in AWS Identity and Access Management Policies". In: *Proceedings of the 2022 on Cloud Computing Security Workshop (CCSW)*. 2022, pp. 63–74

## 5.1 Introduction

Data breaches are a dangerous threat to our society [216]. In 2019, Capital One, an American bank holding, suffered a data breach where data of over a hundred of million people was stolen [82]. More recently, the credit card details of more than hundred million hotel guests were stolen [187], personal data of over 10 million church-goers was leaked [186], and inmate records were leaked from a prison system [144]. All these data breaches have two common characteristics: the data was hosted in the cloud; and the cloud access policies were misconfigured.

Since its introduction in 2006, the use of cloud computing solutions, such as Amazon Simple Storage Service (S3), Microsoft Azure, and Google Cloud Platform increased. While the adoption of cloud services offers a wide range of benefits, it also brings a number of security challenges [44]. As the aforementioned data breaches already demonstrate, security misconfigurations, such as publicly accessible private data, are a considerable threat in cloud security. Attackers even collect publically available lists of misconfigurations to find common vulnerabilities in cloud systems [150]. Moreover, the majority of misconfigurations are reported only when they lead to security incidents [73, 114, 115], making the problem even larger than it first appears. As most cloud environments are exposed to the Internet, misconfigurations create a large attack surface, making it easy for attackers to scan for misconfigured services and exploit them.

To prevent breaches, cloud solutions offer identity and access management (IAM). IAM allows cloud operators to define and manage the roles and access privileges of network users and systems, offloading responsibility to cloud administrators [38, 47]. When configured correctly, IAM systems prevent unauthorized access to protected resources, ensuring that only specified users get access to the specified resources. However, for each newly introduced or modified resource, role or user, these IAM policies must be reconfigured. This (re)configuration is challenging and may unintentionally introduce incorrect rules, which allow access to resources that are undesirable (e.g., guest users accessing sensitive data), potentially leading to security issues. We refer to these undesirable rules as *misconfigurations* and further explain them in Section 5.2. As cloud environments are often large, dynamic, and complex, the configuration of security services becomes difficult and error prone. Therefore, there is a need for systems to flag potential misconfigurations *as soon as* they are introduced and *before* such misconfigurations can be abused.

Existing solutions such as Cloud Custodian [84] use a rule-based approach to prevent the introduction of security misconfigurations in cloud environments. Before deployment, such systems compare IAM policies against existing rules to detect misconfigurations. However, rule-based approaches are limited by the fact that rules need to be created and maintained to adhere to security policies for each specific organization. A rule-based system requires constant effort to keep rules up-to-date, has different requirements per organization, and can be error prone due to manual rule creation. Other approaches, such as P-DIFF [217], monitor access and control behavior to detect misconfigurations. However, this is a reactive approach, meaning that misconfigurations are detected only when they are abused. Unfortunately, at this point, it is often too late, as data is already leaked.

To overcome the limitations of existing solutions, we propose a novel misconfiguration detection approach that is *proactive* in detection, but is able to take into account the *context* of a specific organization and does not require *high maintenance* like rule-based detection. We achieve this by collecting all identity and access management policies from cloud environments before they are rolled out. Next, we model identity and access management policies as a graph where rules are represented by edges between nodes that model entities (users, groups, and roles) and resources. In such graphs, we observed that similar policies within the same environment also have similar graph representations. This means that, correctly configured policies are similar to each other, while misconfigurations show up as anomalies. Leveraging our observation, we apply anomaly detection to spot outliers and raise real-time alerts about potential misconfigurations.

To validate our proposed approach, we collected real-world identity and access management policy data of AWS cloud environments from three different companies. We manually labelled the data as correct policies and potential misconfigurations. On these datasets, our proposed approach correctly detected between 3.7 and 6.4 times as many misconfigurations as rule-based approaches.
In summary, our work makes the following contributions:

- We introduce a novel approach to model AWS identity and access management policies in the form of a graph model;
- We present a novel system that uses anomaly detection techniques to identify potential misconfigurations in Amazon AWS environments;
- We show that our approach correctly detects between 3.7 and 6.4 times as many misconfigurations than state-of-the-art rule-based approaches on a real-world dataset of IAM policies from three AWS cloud environments.

To foster future research on the automatic identification of misconfigurations in cloud environments, we release our prototype open-source: `https://doi.org/10.4121/948f9457-d168-4eb6-9523-bc235a871e83`.

## 5.2 Background and Motivation

We first detail the main concepts used in IAM and then focus on better understanding when access to cloud resources is misconfigured. While we focus on AWS, the concepts discussed here can be applied to any IAM system.

```
{
   "Version":"2012-10-17",
   "name":"AdministratorAccess",
   "Statement":{
      "Effect":"Allow",
      "Action":"*",
      "Resource":"*"
   }
}
```

Figure 5.1: **AdministratorAccess policy that allows all actions (*) on all resources (*).**

**Cloud Identity and Access Management.** Identity and access management allows administrators to limit access to their cloud services and resources. Using IAM, the administrator can create and manage different *entities* that can consist of individual *users*, *groups* of multiple users, and *roles* that can be linked users, groups or even other systems. Each entity has certain *permissions* that allow or deny certain *actions* on cloud resources, such as reading or writing. Sets of permission rules are captured in *policies* that allow administrators a more high level view of related permissions. Cloud providers have different ways of implementing IAM policies, Figure 5.1 gives an example of an AWS' IAM policy [19] in JSON format, which describes an AdministratorAccess policy that allows an administrator to perform any action (*, wildcard) on any resource (*). In this example, the policy itself is related to a *resource*, which means that it still needs to be attached to an entity, i.e. either a user, group, or role. The entity is then granted the permissions specified in the policy.

**Misconfigurations.** While cloud providers often include IAM systems, it is the responsibility of the customer to specify IAM policies for their organization [47]. As organizations differ in their needs, so do the policies they need in place for access and restriction to cloud resources. Keeping a balance between access for maintaining company workflow, and restrictions to improve security is difficult, especially for large organizations where policies often change depending on new needs. This can introduce misconfigurations into IAM policies in several ways. We identify three types of misconfigurations:

1. **Overly permissive policies** allow entities actions on resources that they should not be able to perform.

2. **Overly restrictive policies** denies entities actions on resources that they should be able to perform.

3. **Incorrectly attached policies** specifies correct actions on resources, but for the incorrect entity.

Misconfigured policies open up vulnerabilities that can potentially be exploited by attackers. While there exist some approaches to detect misconfigurations [84, 217], they are either (1) rule-based, requiring a large effort to maintain rules, or (2) reactive, thus detecting misconfigurations only after they are abused. Our approach tackles the following challenges: (i) **Proactive**, misconfigurations need be detected *before* they lead to security incidents; (ii) **Context-specific**, misconfigurations depend on the organization for which they are detected. Some organizations may require stricter or more permissive policies, depending on their workflow; (iii) **Low maintenance**, administrators should not have to maintain rules or perform manual fine-tuning for detecting misconfigurations.

## 5.3 Approach

We aim to detect potential misconfigurations in IAM policies based on the cloud environment for which they are defined. We recall that IAM policies define allowed/disallowed actions on resources and link them as permissions to entities. We propose to model these connections in a graph to represent the connected nature of policies, providing advantages for analyzing and visualizing the policies.

By modelling IAM policies as a graph, we observe that policies permissiveness level is naturally represented through the number of connections (i.e., degree) of each node. Intuitively, an overly permissive policy will have many direct connections to allowed actions, whereas overly restrictive policies will have many direct connections to disallowed actions. We use this observation and leverage anomaly detection techniques [157] to detect misconfigurations. The idea is that we can automatically learn an expected permissiveness level for each resource of an organization given that most policies will be correctly configured. Finally we use these learned models to detect deviations representing overly permissive or restrictive policies and alert operators.

Figure 5.2 shows a high-level overview of the three phases of our approach:

1. **Graph Creation** builds a graph model from the identity and access management policies.

2. **Graph Embedding** extracts relevant features from the graph on which we can apply anomaly detection methods.

3. **Anomaly Detection** trains on past, verified policy data and analyzes new, unverified policies for anomalous misconfigurations. Upon detection we alert an operator for manual verification.

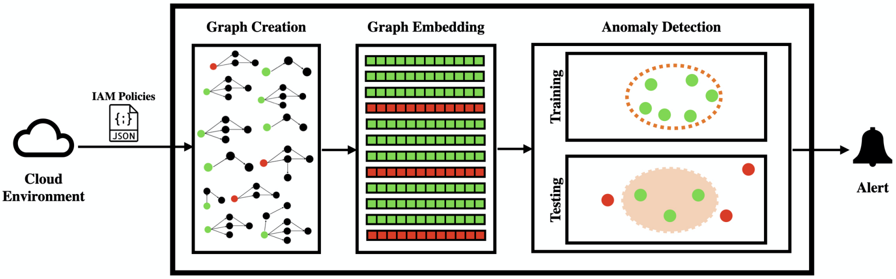


Figure 5.2: **Overview of our proposed approach to detect potential misconfigurations**. First, we create graph model representations of the IAM policies in the cloud environment (Graph Creation); then, we transform the graph model into vector representations (Graph Embedding), and finally, we apply anomaly detection to spot outliers and thereby detect potential misconfigurations.

### 5.3.1 Graph Creation

For the creation of the graph, we transform the policies of a cloud instance into a graph representation. The nodes of the graph represent one of the following types: policy, action, resource, or entity, where entity represents either a user, group, or role. Policy nodes represent the identity and access management policies specified in the environment. Action nodes represent the specified actions within the permissions in the policy, while resource nodes represent the resources on which the actions apply. Entity nodes are the identities that are present within the environment and can make use of the policies. The relationships between the nodes are defined as edges in the graph as follows:

```
(Entity)-[ATTACHED_TO]->(Policy)
(Policy)-[ALLOWS | DENIES]->(Action)
(Action)-[WORKS_ON]->(Resource)
```

### 5.3.2 Graph Embedding

While our graph model accurately represents IAM policies, they cannot directly be used by existing anomaly detection models. This is because anomaly detection models rely on features that indicate normal behavior and find deviations in those features. Therefore, we first transform the graph into a feature representation that anomaly detection models *can* interpret. We recall that intuitively, a large number of allowed actions and low number of disallowed actions attached to a resource may indicate overly permissive policies. Vice versa, a low number of allowed actions and high number of disallowed actions may indicate overly restrictive policies. However, these features may have different values depending on the attached resource, entities or even the type of organization implementing these policies. Therefore, we require method that captures the contextual differences of nodes attached to policies, i.e., type of attached entities, actions and resources. To this end, we use Node2Vec [95], an algorithm that uses random path sampling to create embeddings for each node in the graph [93]. The idea behind Node2Vec is that we select a target node in the graph and perform various random walks, starting from that node. During these walks, we record the types of other nodes we encounter and encode them into a low-dimensional feature space (we use the current best-practice of 128 features). By performing multiple walks for each node in the graph, we capture both information about the number of allowed and disallowed actions, but also of the attached resources and entities. If other nodes have a similar number of allowed and disallowed actions for similar resources and entities, the resulting vector representations will also be similar. We note that there exist other methods that capture similar information, such as graph2vec [151]. However, graph2vec embeds a graph as a whole, making it more difficult to identify which specific policy is misconfigured. As Node2Vec allows us to identify the individual policy node, we argue that it is a better graph embedding for our purpose.

### 5.3.3 Anomaly Detection

Now that we have embedded policy nodes of the graph, we can use anomaly detection methods to identify potential misconfigurations. We recall that similar policies will also have similar graph representations. Therefore, embeddings of properly configured policies will be similar to each other, and misconfigurations will be different. Anomaly detection techniques will be able to spot these outliers and thereby detect potential misconfigurations.

The anomaly detection model should ideally be trained on properly configured policies (we discuss misconfigurations in training data in Section 5.6.3). These can be policies that follow security policies, adhere to industry best practices, or are created through the use of existing tools to ensure proper configuration. Then, when policies change or when new policies are introduced, the anomaly detection model checks them for outliers and marks them as potential misconfigurations. This continuous monitoring of changes in cloud environments enables our approach to detect potential misconfigurations immediately when there is a change in policies. This quick reaction time minimizes the time a cloud environment is exposed, and therefore minimizes the possibility for abuse.

Based on our empirical evaluation in Section 5.6, we use Local Outlier Factor [43] (LOF) as our anomaly detection model. LOF is a density-based model, where a data point is considered an outlier if it has a lower local density compared to its neighbors. The model does not make any assumption on the probability distribution of the data and it provides some tolerance in case of outliers in the training set, making it a generic solution suitable for our approach.

## 5.4 Implementation

While we described our approach as a generic cloud IAM misconfiguration detection system, our implementation focuses on the AWS cloud platform, as it is currently the largest cloud provider [195].

**Graph Creation.** To create our graph, we pull the IAM policies from the monitored AWS environment in JSON format and transform the file to a Neo4j graph database [94]. We recall from Section 5.2 and Figure 5.1 that policies contain multiple statements that specify actions and resources. We create nodes for each policy, action and resource in the JSON file, and connect them with edges according to their relationship defined in the statements. We perform a similar operation for the entities defined in the JSON file and connect them to their corresponding policies. Figure 5.3 shows an example of a resulting graph, where yellow nodes represents the policy names, blue nodes represent actions, and purple nodes represent resources. Existing graphs can easily be updated if changes are made to the policies in the environment. We define a change as adding, removing or modifying nodes, edges or any of their properties. Changes in policies can be automatically detected from differing lines in IAM JSON files and can be applied as updates to the graph. This mechanism updates the graph rather than recreating it entirely, making it more efficient (see Section 5.6.4). Moreover, because the updates are triggered automatically, there is minimal overhead for security operators as they only have to focus on verifying detected misconfigurations.

**Graph Embedding.** Next, we transform a graph to vectors representing policies which we can then use in anomaly detection models. We run the algorithm Node2vec [95] included in Neo4j on each policy node in the graph to produce an embedded vector of 128 features (128 features are currently considered best practice). We then store the resulting embedded feature vectors as properties in the policy nodes. This way, we can access the embedded vectors by querying the database, enabling us to easily extract the vectors for anomaly detection.

**Anomaly Detection.** Finally, we use the scikit-learn Python library [162]. This library implements a wide range of machine learning algorithms, including the Local Outlier Factor algorithm and other algorithms used in our evaluation. More details on the implementation of the anomaly detection will be discussed in Section 5.6.

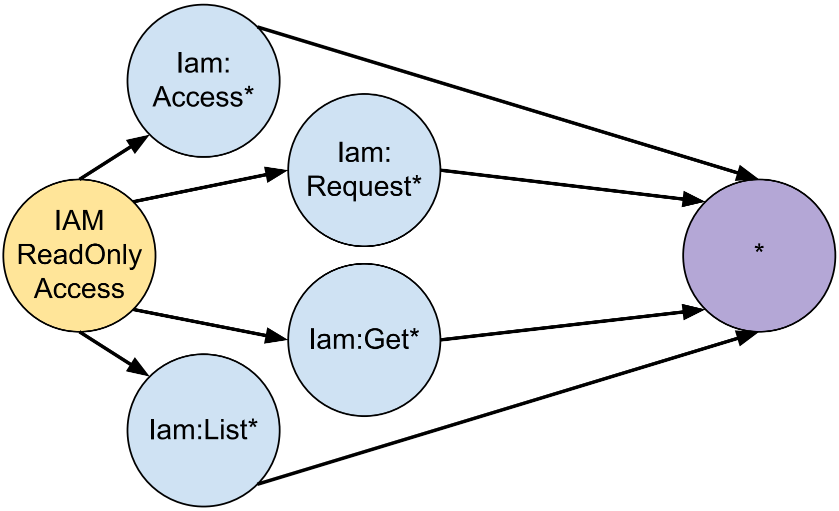


Figure 5.3: **Example of a graph representation of a policy in the environment.** Policies (yellow) allow actions (blue) on resources (purple).

## 5.5 Dataset

To evaluate our prototype, we need real IAM policies containing labelled misconfigurations. We collected a total of 2480 different policies from three organizations that use AWS cloud environments over a period of several weeks. These datasets are detailed in Table 5.1. The first two datasets were collected from two different financial organizations with approximately 12,000 (dataset 1) and 130 (dataset 2) employees. The third dataset belongs to a smaller tech/software development company with 4 employees. These datasets represent small, medium and large enterprises, giving us a variety of real-world deployments. While this dataset is relatively small (three organizations), we use all the deployed policies from real-world environments, minimizing potential sampling bias. A further advantage is that we have a realistic ratio of misconfigurations (which are relatively scarce), minimizing any base rate fallacy.

We collected this data with a script that uses the AWS CLI [31] to periodically pull IAM policies from the cloud environments of these organizations[1]. We collected all IAM policies, as well as all entities (user, groups, and roles) from each environment and store it locally in a spreadsheet. Please note that collecting AWS IAM policies can contain sensitive data about the resources used by an organization and should therefore be treated confidently or be sanitized (e.g., replacing ARNs with generic identifiers) when shared with third parties.

As shown in Table 5.1, the organizations implemented between 812 and 842 poli-

[1] Code available at `https://doi.org/10.4121/948f9457-d168-4eb6-9523-bc235a871e83`

Table 5.1: **Overview of three datasets used for the evaluation.** We show the size of the datasets in terms of number of employees working for each organization as well as the number of policies, users, groups, roles defined in the IAM policies. To evaluate changes in datasets, we collected policies multiple times over a period of multiple weeks indicated by *Number of collections*.

| | **Total number of** | | | | | **Number of** |
|---|---|---|---|---|---|---|
| **Dataset** | **employees** | **policies** | **users** | **groups** | **roles** | **collections** |
| 1 | 12,000 | 842 | 0 | 0 | 55 | 8 |
| 2 | 130 | 812 | 0 | 0 | 34 | 2 |
| 3 | 4 | 826 | 2 | 1 | 10 | 12 |

cies. This relatively high number of policies is due to the fact that all companies used the 515 default IAM policies provided by AWS themselves. Each company added between 297 and 327 custom policies for their environment. The first two datasets do not contain any users or user groups because those environments authenticate users with a separate identity provider that issues temporary cloud credentials. This automatically ensures users assume roles rather than having permissions on their own user account, a common industry best-practice. We note that this does not impact our approach as we treat users, groups and roles all as entities. Furthermore, we collected IAM policies of each organization at regular intervals during a period of multiple weeks (at least two weeks, as shown in Table 5.1). This allows us to evaluate the performance of our approach when misconfigurations are introduced after the initial configuration.

**Data Labeling.** To evaluate our proposed approach, the collected policies were manually labelled into correct and misconfigured policies. Each policy has been manually reviewed with respect to the level of permissiveness. Policies that contain a high number of allowed actions, on a high number of resources, were considered with extra care. Each policy has been compared against current industry best practices [29], and, by following these guidelines, we attempted to minimize label inaccuracy. An example of a potential misconfiguration is the previously mentioned AdministratorAccess policy (see Figure 5.1). This policy has a high level of permissiveness since it grants permission to perform all the actions on all the resources. When not attached to the proper entities this policy could be a potential misconfiguration. All three datasets were labeled, and contain 12, 11, and 6 misconfigurations, respectively. During our evaluation, we analyze misconfigurations that were introduced after initial configuration. However, our labelling process showed that no new misconfigurations were introduced in the modifications. Therefore, in those sections, we simulate the temporal aspect by manually introducing misconfigurations after the initial setup.

## 5.6 Evaluation

To evaluate our prototype, we take care to follow good practices in machine learning detailed by Arp et al. [26] and TESSERACT [163]. In each experiment, we briefly discuss how we tried to avoid relevant common pitfalls.

We evaluate our prototype by comparing it against Cloud Custodian [84], one of the main rule-based tools used for validating many AWS IAM policies, which can be seen as our experimental baseline. Cloud Custodian allows operators to define custom rules to ensure deployed IAM policies adhere to organizational requirements. The organizations represented in our dataset did not have any predefined rules in place. Thus, we instead compared our approach against Cloud Custodian using open-source IAM rules[2]. We note that open-source rules such as those used for this experiment are not tailored towards a specific setting and will therefore likely not give the best possible results. To make the comparison more fair, we also performed the same experiment where we manually selected open-source rules that are applicable to the organizations in our dataset, removing rules producing incorrect detections. This emphasizes the fundamental limitation of rule-based systems, i.e., there are no generic rules that can be applied to all cloud environments. Hence, an anomaly-based solution like ours can help reduce the manual effort required for detecting IAM misconfigurations while potentially detecting more misconfigurations.

Table 5.2 shows the performance of both our approach using the LOF anomaly detector (see Section 5.6.1), and the two rulesets of Cloud Custodian on all three datasets split randomly into 90/10 training and testing sets. While Cloud Custodian performs better over all policies, we find that for misconfigurations, our approach performs better, correctly detecting between 3.7 and 6.4 times as many misconfigurations. The reason for this is that rule-based systems such as Cloud Custodian are very precise in their detection, i.e. if there exists a rule for a misconfiguration, it will only trigger for the misconfiguration and not for other rules. Our approach on the other hand is anomaly-based, meaning that it may incorrectly flag some correct configurations as misconfigurations, but it is also able to detect misconfigurations not captured by rules. Therefore, our approach shows a promising direction for detecting additional misconfigurations using anomaly detection, improving the overall cloud security.

**Misclassifications.** During our evaluation, we found some occurences of false positives (correct policies flagged as misconfigurations) and false negatives (unde-

[2] `https://github.com/davidclin/cloudcustodian-policies`

Table 5.2: **Overall evaluation.** Performance of our approach compared with Cloud Custodian using all rules, and using rules specifically selected for our dataset. We show the performance for detecting misconfigurations (top) and the overall performance (bottom). While Cloud Custodian seems to perform better overall, its recall is low, meaning the majority of misconfigurations go undetected.

| | | **Our approach** | | | **Cloud Custodian** | | | **Cloud Custodian** | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | **All rules** | | | **Selected rules** | | |
| | **DS** | **Prec.** | **Recall** | **F1-score** | **Prec.** | **Recall** | **F1-score** | **Prec.** | **Recall** | **F1-score** |
| Misconf. | 1 | 66.67% | 66.67% | 66.67% | 7.89% | 10.34% | 4.48% | 100.00% | 10.34% | 9.37% |
| | 2 | 70.00% | 63.34% | 66.67% | 13.73% | 17.07% | 7.61% | 100.00% | 17.07% | 14.58% |
| | 3 | 75.00% | 50.00% | 60.00% | 15.38% | 11.32% | 6.52% | 100.00% | 11.32% | 10.17% |
| Overall | 1 | 91.58% | 91.58% | 91.58% | 97.93% | 97.60% | 97.76% | 98.99% | 98.98% | 98.57% |
| | 2 | 92.03% | 92.31% | 92.15% | 97.40% | 97.09% | 97.24% | 98.75% | 98.73% | 98.28% |
| | 3 | 94.97% | 95.45% | 95.03% | 98.93% | 97.88% | 96.87% | 98.12% | 98.08% | 97.33% |

tected misconfigurations). The common characteristics of our false positives are high levels of permissiveness, which in many scenarios are misconfigurations, but under certain circumstances can be allowed. An example of such policy is the *Read-Only* policy, which, in dataset 1 permits 762 read-only actions, making it very permissive. However, since in this dataset, all read-only permissions are for on non-critical resources the policy was not considered a misconfiguration. As our anomaly detector does not have a sense of which resources are critical or not, our approach plays it safe and classifies these policies as potential misconfigurations.

Conversely, false negatives can occur when a policy does not seem permissive but actually allows certain high-impact actions. A policy such as the *PowerUser-Access* allows only a small number of actions, for a limited number of entities, but works on critical resources and can have a high impact. Therefore, to reduce the number of misclassifications, further research is needed into methods that take into account the impact of policies.

### 5.6.1 Anomaly Detectors

We have shown that our approach correctly detects more misconfigurations than rule-based approaches using LOF as our anomaly detector. The only requirement for our detector is that it can detect anomalies based on a vector representation. Therefore, before choosing LOF, we compared four different anomaly detection techniques to empirically find the best performing technique for our approach: 1) One-Class Support Vector Machine; 2) Local Outlier Factor; 3) Isolation Forest; 4) Robust Covariance.

For this experiment, we limit ourselves to dataset 1 to minimize the data snooping bias that the result of this experiment may introduce to our overall evaluation. Ideally, such evaluation would be performed on a separate dataset to exclude bias completely. However, due the limited availability of data we chose to still report the performance of dataset 1 in Table 5.2, but show the results separately for each dataset, demonstrating that the performance generalizes across datasets. We follow our proposed approach as described in Section 5.3. First, we create a graph model representation from the first collected IAM policies. Second, we apply node2vec to embed the policy nodes into vector representations. The vector representations are the same for all four tested anomaly detectors. Next, we split dataset 1 into a 90/10 training and testing set, following machine learning best practices [26, 163]. Dataset 1 contains a total of 842 policies containing 12 misconfigurations. To ensure that the anomaly detection model is solely trained on correctly configured policies, we temporarily remove the misconfigurations from the dataset, leaving us with 830 correct policies. Using a 90/10 split, we create a training set consisting of 747 (90%) correct policies without any misconfiguration. The test set consists of the remaining 83 (10%) correct policies, and the 12 misconfigurations, for a total test size of 95 policies. We choose a 90/10 training testing split as IAM policy changes are often relatively small compared to the existing policies. In these environments, the majority of policies are created initially when the environment is setup. New policies are added through the life-cycle of the environment, but comprise a smaller part of the total amount. Additionally, the test set is imbalanced as there are considerably more correct policies than misconfigurations which is also representative of real-world deployments, since misconfigurations are introduced much less frequently than correct modifications [26, 163]. In this setting, the training set can be seen as the initial deployment of a cloud environment. The test set then simulates the changes made to the policies in the environment, both additions of policies as well as modifications, and can therefore be considered new observations.

Each anomaly detector goes through the same process outlined in Section 5.3.3. We first train the model using the correct training policies. Then, we evaluate the performance of the model using the test set, containing both correct policies and misconfigurations. We evaluate the performance using the following metrics: precision, recall, F1-score, and ROC Area Under Curve (ROC AUC). The precision measures the proportion of correctly identified misconfigurations in all detected misconfigurations. The recall measures the proportion of correctly identified misconfigurations in all actual misconfigurations. The F1-score is the harmonic mean of precision and recall, and conveys the balance between the precision and the recall. The ROC AUC measures the relationship between the True Positive and False Positive Rate of the anomaly detector.

**Parameter optimization.** The performance of each anomaly detector depends on the use of specific parameters that determine how each algorithm separates misconfigurations from valid configurations. Therefore, we performed a grid-search on dataset 1 to find the optimal parameters for each of the four anomaly detection algorithms. The effect of the parameters on the performance metrics can be found in Figure 5.4. For each parameter, we choose the best performing according to the ROC AUC metric as a high ROC AUC will not only give a high precision, but will also be robust against variations in the data. Using our grid-search, we found the following optimal parameters for each anomaly detector:

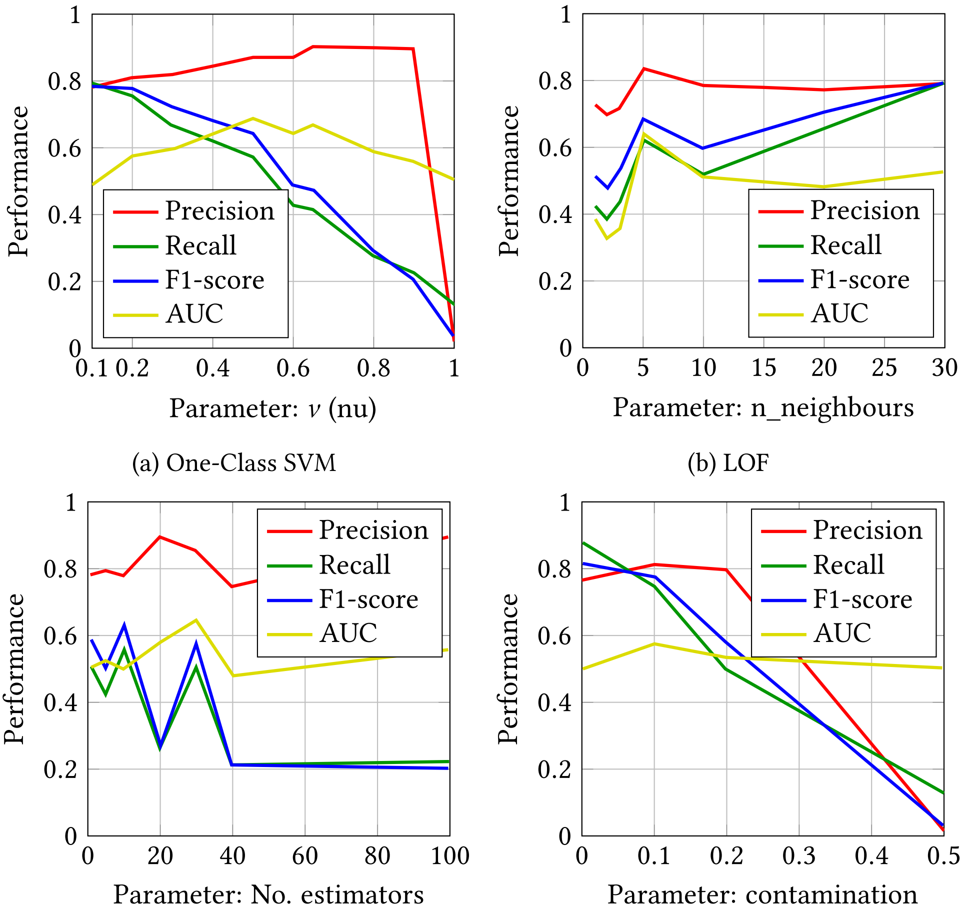
- One-Class SVM: gamma = 0.001, nu = 0.5
- Local Outlier Factor: n_neighbours = 5
- Isolation Forest: n_estimators = 30
- Robust Covariance: contamination = 0.1

**Results.** With the aforementioned selected optimal parameters, we evaluate the performances of the four anomaly detection techniques. Table 5.3 shows the obtained overall performance for each anomaly detector for their optimal parameters. Our first observation is that all four techniques have relatively high precision and recall. Both metrics are important since precision measures whether detected misconfigurations are *actual* misconfigurations, while recall measures the proportion of *detected* misconfigurations. In our approach, however, we prefer a high recall above a high precision because potentially missed misconfigurations have a considerably larger impact to cloud security than a false positive detection, which can be manually filtered out by an operator.

From the results in Table 5.3 we find that the Local Outlier Factor (LOF) has the best performance in precision, recall and F1-score. The ROC AUC of the One-Class SVM is slightly higher, meaning that the choice of decision boundary is slightly more robust and will therefore depend less on the chosen parameters. Nevertheless, the ROC AUC of the LOF algorithm is the second highest and gives a better overall performance, making it our algorithm of choice. Additionally, LOF has the great advantage of being more resilient in cases where the training set contains anomalous data points (see Section 5.6.3). In fact, by measuring the local deviation of data points with respect to their neighbors, LOF can identify local outliers in the training set, providing some tolerance for misconfigurations already present during the training.

### 5.6.2 Parameter Transferability

Our approach shows promising results for detecting more misconfigurations than rule-based approaches. However, with large manual effort, rules may be tweaked



(a) One-Class SVM

(b) LOF

(c) Isolation Forest

(d) Robust Covariance

Figure 5.4: **Parameter selection.** Detection performance metrics during parameter optimization for the four anomaly detection techniques

to the extent that they cover nearly every edge case. Therefore, we want to evaluate whether our anomaly-based approach requires a similar level of tweaking, or whether our internal parameters are transferable accross settings. To this end, we take our optimal LOF parameter (n_neighbours = 5) for dataset 1 as found in Section 5.6.1 and apply them to datasets 2 and 3 to see how well they generalize.

Analogous to our approach, we create separate graph models for the two new datasets and embed the policy nodes into vector representations. We then apply the same 90/10 training testing split as in the previous experiment and obtain 737 correct policies for training and 11 misconfigurations and 64 correct policies for testing dataset 2. For dataset 3, the training set contains 743 correct policies and the

Table 5.3: **Comparison of anomaly detectors.** Performance of anomaly detectors after parameter optimization (dataset 1).

| Algorithm | Precision | Recall | F1-Score | AUC |
|---|---|---|---|---|
| One-Class SVM | 88.78% | 89.47% | 89.06% | 0.70 |
| Local Outlier Factor | 91.58% | 91.58% | 91.58% | 0.66 |
| Isolation Forest | 84.08% | 87.37% | 84.50% | 0.60 |
| Robust Covariance | 87.94% | 89.47% | 87.90% | 0.65 |

testing set contains 6 misconfigurations, and 77 correct policies. We create separate LOF models for the training sets, using the n_neighbours = 5 parameter and evaluate the performance using the test sets.

Table 5.4 shows the results of this experiment. We observe that all metrics are quite similar compared to the baseline, and even slightly improved. The reason that parameters are transferable is the relatively high similarity between AWS IAM policies. By default, there are 515 IAM policies managed and provided by AWS, meaning that there is a significant overlap (> 60%) of policies between the datasets. As these default policies are well-checked, our model is trained on properly configured policies that are available in nearly all real-world scenarios. The small variance between our baseline and the other datasets can be explained by the level of permissiveness for policies in the dataset. In dataset 1, policies were more permissive, without being classified as misconfigurations, meaning it is more difficult to distinct between correct and misconfigured policies. In conclusion, we find that that the parameter, n_neighbours = 5, seems to be transferable between datasets.

### 5.6.3 Misconfigurations in Training Data

As our approach is based on anomaly detection, misconfigurations in training data may lead to undetected misconfigurations during deployment. To test the resilience against misconfigurations in the training data, we introduced known misconfigu-

Table 5.4: **Parameter transferability.** LOF detection performance on dataset 2 and 3, with parameter n_neighbours = 5.

| Dataset | Precision | Recall | F1-Score | ROC AUC |
|---|---|---|---|---|
| 1 (baseline) | 91.58% | 91.58% | 91.58% | 0.66 |
| 2 | 92.03% | 92.31% | 92.15% | 0.73 |
| 3 | 94.97% | 95.45% | 95.03% | 0.72 |

Table 5.5: **Misconfigurations in training data.** We show the influence of including a certain number of misconfigurations in the training dataset.

| Algorithm | No. Misconfigurations | Precision | Recall | F1-score |
|---|---|---|---|---|
| One-class SVM | 1 ( 0.12% of training) | 90.68% | 50.05% | 63.15% |
| One-class SVM | 5 ( 0.61% of training) | 90.81% | 48.20% | 61.83% |
| One-class SVM | 10 ( 1.21% of training) | 92.09% | 49.71% | 63.49% |
| Local Outlier Factor | 1 ( 0.12% of training) | 91.07% | 95.24% | 93.10% |
| Local Outlier Factor | 5 ( 0.61% of training) | 91.78% | 95.61% | 93.65% |
| Local Outlier Factor | 10 ( 1.21% of training) | 92.67% | 95.78% | 94.20% |
| Isolation Forest | 1 ( 0.12% of training) | 88.67% | 17.10% | 23.94% |
| Isolation Forest | 5 ( 0.61% of training) | 91.08% | 28.47% | 39.81% |
| Isolation Forest | 10 ( 1.21% of training) | 91.86% | 30.10% | 42.84% |
| Robust Covariance | 1 ( 0.12% of training) | 90.64% | 81.73% | 85.91% |
| Robust Covariance | 5 ( 0.61% of training) | 91.57% | 85.07% | 88.17% |
| Robust Covariance | 10 ( 1.21% of training) | 92.37% | 82.65% | 87.19% |

rations during the training phase of all four algorithms and tested the performance with respect to the amount of introduced misconfigurations.

Table 5.5 shows the result of this experiment. We see that for the One-class SVM and Isolation Forest, the recall drops significantly with respect to the data in Table 5.3. The Local Outlier Factor and Robust Covariance algorithms are much more resilient against misconfigurations in the training data. We also note that increasing the number of misconfigurations seems to increase the precision, recall and F1-score. However, this is due to fewer misconfigurations in the test data as they were included in the training data instead. From Table 5.2, we found that evaluation metrics over misconfigurations are relatively lower, hence having fewer misconfigurations slightly skews the evaluation upwards. Nevertheless, our experiments show that both the Local Outlier Factor and Robust Covariance algorithms are relatively robust against misconfigurations in the training data.

### 5.6.4 Runtime Performance

During our experiments, we measured the runtime performance of the graph creation and model training stage of our approach for various input sizes. All experiments were performed on an Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz laptop running Ubuntu 20.04 LTS. We report the average runtime over 10 runs.

**Graph Creation.** During Graph Creation, the IAM policy data is transformed into a graph model representation. Figure 5.5 shows the runtime performance of the Graph Creation stage. The performance scales linearly, but larger datasets can take up a significant amount of time. For our largest dataset (1) with 842 policies, graph creation took 20 minutes and 55 seconds. It is worth mentioning that only the initial graph creation is this long, due to the fact that all nodes must be created. Graph updates are considerably faster since the number of new nodes is generally low. To evaluate this, we performed 10 experiments where we took the graph of Dataset 3 consisting of 826 nodes. Next, we added 10 random policies and added these to the graph, which took on average 0.581 seconds versus 833.986 seconds for recreating the graph. This shows that our graph update achieved a speedup of a factor 1091 versus recreating the graph from scratch. While the speedup depends on both the total size of the graph (influencing the recreation of the graph) and the number of added nodes (influencing the average update time), we believe our experiment shows a considerable speedup in a realistic scenario.

**Graph Embedding.** After the graph has been created, we run the Node2vec algorithm on the policy nodes to transform nodes into vectors, which took on average 57 milliseconds for our entire dataset.

**Model Training and Prediction.** We also consider the model training overhead of the Local Outlier Factor anomaly detection model. The LOF model is trained on correct policies and then used to determine whether new observations are also correct or potential misconfigurations. The measured runtime performance of the LOF model can be found in Figure 5.6. We find that the runtime performance is negligible with respect to creating the graph model with 26 milliseconds for our largest dataset. For the prediction, we measured an average of 6 milliseconds for our largest test dataset of 86 policies and is therefore also negligible.

## 5.7 Discussion and Future Work

**Challenges.** We recall from Section 5.2 that our approach aims to be **proactive**, **context-specific**, and **low maintenance**. Our approach can be automatically triggered upon changes in IAM policies as explained in Section 5.4. Furthermore, our evaluation in Section 5.6.4 has shown that our approach can also be run in a reasonable timespan, allowing for **proactive** misconfiguration detection. Additionally, our evaluation in Table 5.2 has shown that our anomaly-based approach detects more misconfigurations, compared to generic rule-based approaches, meaning it is better able to take into account the **context-specific** (mis)configurations for an organization. And finaly, with our Parameter Transferability evaluation in Section 5.6.2, we

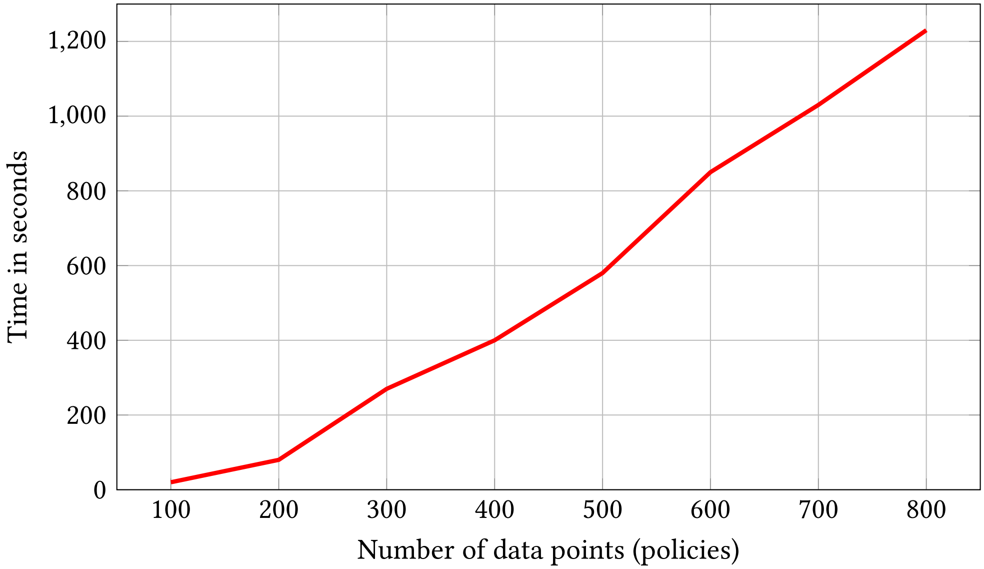


Figure 5.5: **Overview of runtime performance for the creation of the graph model.**

have shown that our approach is **low maintenance**, showing promising results for all our criterea.

**Misconfigurations in training data.** Our approach assumed that we only train the anomaly detection model on correct policies. While our evaluation in Section 5.6.3 showed that Local Outlier Factor and Robust Covariance have some robustness with small numbers of misconfigurations, ideally the training data should not contain misconfigurations. The policy data collected for our experiments is from real-world cloud environments. Each dataset has been manually reviewed, and potential misconfigurations have been removed from the training set. This is a costly human operation that is error prone and may not be feasible in all use cases. Therefore, a possible approach could be to train only on the 515 default IAM policies provided by AWS. In this scenario, we can be sure that there are no misconfigurations in the training data. As a disadvantage, our approach will not be able to learn the context-specific environment and may flag policies that seem misconfigured with respect to IAM policies, but are permissible in certain cases. In short, we believe that with careful review, the number of misconfigurations that slip through manual detection is sufficiently low, and we have shown that for low numbers of misconfigurations, Local Outlier Factor and Robust Covariance still produce good results.

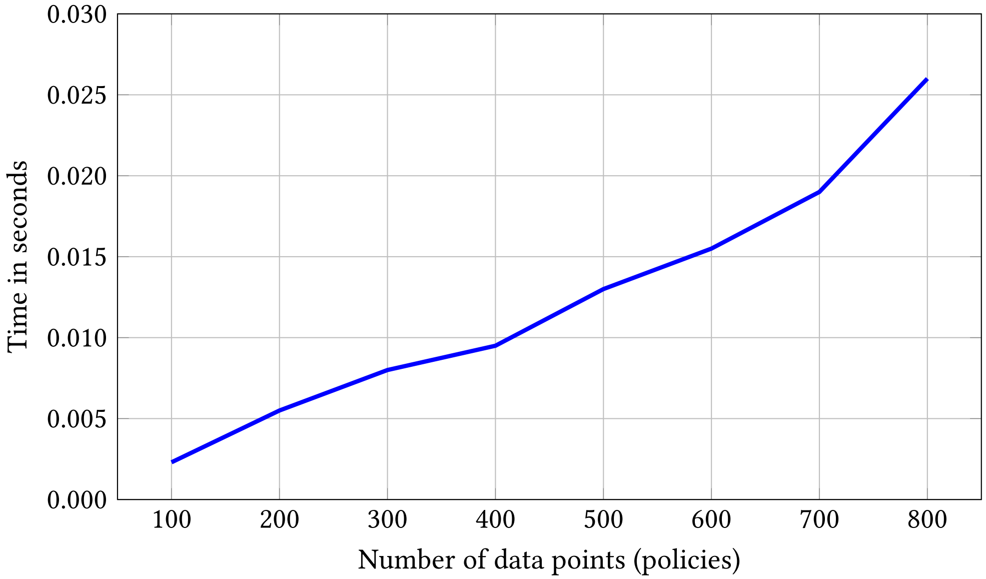


Figure 5.6: **Overview of runtime performance for training the LOF anomaly detector.**

**Advanced embedding and anomaly detection.** In our approach, we have used the graph embedding technique Node2vec, which is currently the state-of-the-art of graph embedding. Node2vec in combination with LOF has already provided us with good results in detecting potential misconfigurations. There are however newer techniques in the making that might be able to transform the graph in a better and more efficient way. An example of such a new embedding technique is GraphSage [97], which uses inductive representation learning to also enable the embedding of node properties. Furthermore, we have only considered four anomaly detection algorithms in our approach. More complex machine learning techniques could further enhance the performance of our approach. Examples of such techniques are Graph Convolutional Networks [116], and One-Class Neural Networks [49].

**Different policy types.** In our current approach, we only consider identity-based policies. There are, however, more IAM policie types in cloud environments[3]:

- Resource-based policies,
- Permission boundaries,
- Organization service contol policies,

[3] https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html

- Access control lists,
- Session policies.

These policies cover different scenarios and can be represented using other node types in a graph. E.g., a resource-based policy can set a storage service node to be either publicly or privately accessible. Such policies specify actions that are only allowed on that specific resource, regardless of who is performing the action. These different policy types can be created and stored in the same way as identity-based policies but may show different behaviour when analyzed using our approach. Therefore, we recommend future investigation into using our approach with different policy types.

**Future Work.** From our current approach, we see several potential future directions for improvement. First, a feedback loop may be added after a security operator verifies or discards flagged misconfigurations. The model can be updated with such new information and prevent other similar alerts in the future.

Second, our work focused on the AWS cloud platform. While we only evaluated our approach on AWS IAM datasets, the technique itself may be extended to other cloud platforms that provide IAM policies where entities can be linked to resources through policies. Besides AWS, other major cloud platforms such as Google Cloud Platform and Microsoft Azure support such policy structures. Our approach can be extended by adding collection services for additional cloud providers. While the rest of our approach should be independent of the cloud provider, future research should show whether our approach achieves similar results on those platforms.

Finally, our approach is able to detect misconfigurations, but is not yet able to provide suggestsions for IAM policy modifications. To this end, we could explore link prediction techniques that indicate which links should be added or removed. While link prediction is technically possible to implement in our graph, we consider this out of scope for the current research.

## 5.8 Related Work

Access control is a subfield of the broader area of identity and access management, and has been studied extensively. P-Diff [217] monitors access and control behavior by using decision tree algorithms. While effective, P-Diff has one major limitation: it learns access control policies from access logs. This means that detection is therefore limited to the information contained in the access logs, which makes the approach reactive. As a consequence, misconfigurations can only be detected after abuse, once anomalies show up in the access logs. Instead, our approach is proactive and aims at identifying misconfigurations as soon as they are introduced in the cloud environment. Baaz [66] infers permission misconfigurations in an enterprise network by monitoring updates made to the access control metadata, and looking for potential inconsistencies among peers. The major limitation of Baaz is that it relies on the definition of what should be considered as an inconsistency. This parameter can be tweaked by administrators, but could still cause problems and influence the performance of the system.

Rule-based solutions rely on predefined rules to which the newly created or modified cloud resources must adhere—these rules have to be created, monitored and maintained throughout the life cycle of the cloud environment. Cloud Custodian [84] is a widely used open-source rule-based system. Cloud Custodian enables users to be well managed in the cloud. It allows for the easy definition of rules to manage the cloud infrastructure. These rules are collected in policies. The policies can be as simple or as complex as the person creating them wants them to be. Examples of such policies can be the blocking of all the public access to S3 buckets or the detection of an account receiving admin privileges. AWS Remediation Framework [81] is another example of an open-source solution. As the name suggests, it is a project that identifies and remediates AWS security issues to ensure AWS usage is compliant with a set of rules. Although these rule-based solutions can be very powerful and have clear advantages, there are also a number of limitations. First of all, the rules need to be created and maintained to adhere to security policies. This has to be performed manually and can require a large effort. Furthermore, this process can be error prone and security policies can be insufficient to detect all misconfigurations. Secondly, cloud environments are generally extremely dynamic and change frequently. There are situations in which a certain action can be seen as a misconfiguration, while it is needed for a certain operation, predefined rules can therefore be too rigid too handle these quick changes, which will impact the performance of the system.

Cloud providers have also started offering solutions for detecting misconfigurations. AWS provides CloudTrail [30], which is an AWS service that enables governance, compliance, and auditing of the AWS account and all the corresponding resources. It provides logging and continuous monitoring of the AWS environment. Cloudtrail can be used in two ways to detect misconfigurations. First, it can be used to log and raise alerts in case any changes are made to the identity and access management configurations of the cloud resources. Second, it can be used to detect unauthorized access if misconfigurations are abused. Both ways have limitations. Either over-alerting administrators on every change made, or reacting to already happened abuse, thus being too late. Besides, AWS has some mechanisms in place to prevent misconfigurations. For example, when overly permissive identity and access management roles are created, the system raises a warning. This already creates a first line of defense, however, it can be easily overridden by administrators and only identifies major and obvious errors.

Other studies, which are orthogonal to our research, focused on measuring misconfigurations. Continella et al. [63] investigated permission misconfigurations on Amazon Simple Storage Service (S3) buckets [28]. Another research has been performed on the cause of data leaks when cloud platforms are used as mobile app back-ends [228]. Finally, Zahoor et al. developed a formal method for detecting conflicting policies [223] and extended this method to work accross multiple cloud providers [224]. These conflicts cover cases where some policies allow entities access to a resource, whereas another policies deny overlapping entities from accessing to same resource. While such policies are also misconfigurations, they are orthogonal to the context-specific overly permissive or restrictive policies that our work focuses on.

## 5.9 Conclusion

In this chapter, we presented a novel approach for detecting misconfigurations of AWS identity and access management policies. The goals for this approach were to be proactive, context-specific, and requiring low maintenance. To achieve these goals, we first created a graph model representing IAM policies from a given cloud environment. We then created context-specific representations of all policies using node2vec embeddings. Finally, we trained an anomaly detection model on correct policy embeddings and used it to detect potential misconfigurations in new policies. We have evaluated our approach on real-world IAM policies from three organizations and have shown that our approach correctly detects between 3.7 and 6.4 times more misconfigurations than state-of-the-art approaches at the cost of a slight decrease in precision. Furthermore, we have shown that the parameters for the anomaly detection algorithms are transferable between environments, while still maintaining a similar detection performance, ensuring low maintenance costs. As security misconfigurations in cloud environments have detrimental consequences, our approach performs an important step to reduce the risk of security misconfigurations.

# Chapter 6

## Concluding Remarks

We have seen that adversaries continue to target organizations using increasingly advanced strategies to execute their attacks, often involving multi-step kill chains. While security operators try to uncover these kill chains, they base their detection on security events that provide only an isolated view of single steps within such an attack. Moreover, the number of security events produced to detect attacks is overwhelming and often includes events that are not malicious. This leads to operators suffering from alert fatigue, making it difficult to correctly identify kill chains. To combat these problems and gain a deeper understanding of security events, we proposed a context-based approach to answer our main research question:

**Main RQ.** *To what extent can we leverage high-level contextual knowledge of cyber kill chains into security event analysis?*

This chapter summarizes our contributions, reflects on the thesis process, addresses some of our limitations and provides insights into future work.

## 6.1 Contributions

To answer our main research question, we focused on three open problems in the current workflow of SOC operators and performed one case study. Figure 6.1 shows how these sub-problems fit into the overall process of dealing with security events.

We started by investigating the high-level CTI reports discussing Tactics, Techniques and Procedures that operators share to spread knowledge about cyber kill

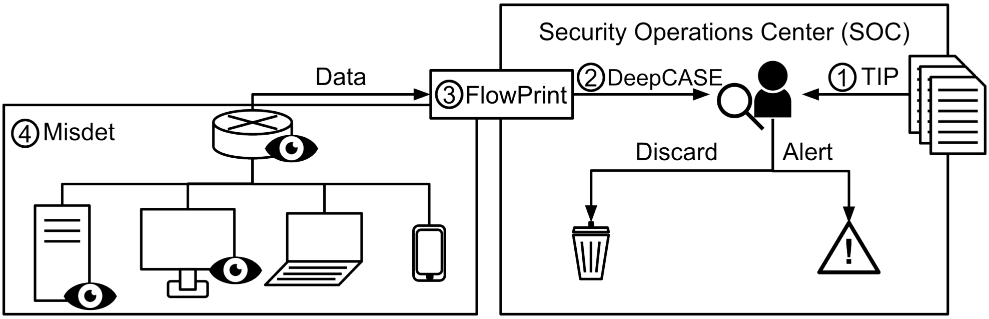


Figure 6.1: **Thesis Overview.** Four focal points of context-based identification and explanation of security events.

chains. We introduced EAGLE, our approach for Threat Intelligence Processing to answer our first concrete research question:

> **RQ 1.** *To what extent can we automate knowledge extraction from CTI reports and classify it into existing TTP frameworks?*

We found that CTI reports use widely different descriptions to communicate similar tactics, techniques and procedures. Moreover, existing techniques to detect MITRE ATT&CK concepts mentioned in CTI reports do not follow well-established NLP techniques and therefore often fail to detect a large portion of concepts. To solve this problem, we introduced the EAGLE framework that combines established NLP techniques with domain-specific knowledge of cyber attacks. By doing so, we created an extendable approach that not only detects ATT&CK concepts, but can also infer relations between concepts, leading to a more complete overview of TTP information contained in CTI reports. Moreover, by combining semantic and data-driven approaches, we showed how to increase detection performance, while still making the detected results explainable to security operators. This process greatly increases the effectiveness of security operators when searching for relevant threat intelligence and linking intelligence to observed security events.

Next, we shifted our focus from external knowledge about threat intelligence to the actual security events that are observed within a SOC. Here we found that the knowledge from CTI reports was used by security operators to assess security events. However, not all attacks may be covered by external knowledge, especially in the case of newly emerging threats. Moreover, even if knowledge is available, the process of triaging (i.e., the initial decision of whether security events are malicious or benign) is a time-consuming task. Therefore, with our next research, we tried to answer our second research question:

> **RQ 2.** *To what extent can we reduce the manual workload of triaging security events based on contextual information?*

To answer RQ 2, we developed our approach DeepCASE, which assists security operators in identifying kill chains by analyzing the context in which events occur. Our main observation was that security events that are part of an actual kill chain, do not occur as a single event, but are highly correlated with other security events that represent different parts of the same kill chain. To leverage this insight, we created a deep learning-based approach that attempts to find correlations between security events from large sequences of real-world events. Once these correlations are found, we cluster similar sequences of correlated events and show these clusters to security operators who decide whether found sequences are malicious or whether they are benign correlations. Previous decisions can be repeated for new, similar sequences of events, thereby vastly reducing the workload of security operators.

For our third question, we challenged a fundamental assumption of our work, namely that enough contextual information is provided to security operators to decide whether a security event is malicious or benign. We found that host-based approaches often provide additional information about processes running on the device. However, in network-based detection, this information is often lacking. Therefore, we focused on the following research question:

> **RQ 3.** *To what extent can we identify malicious applications based on network traffic?*

For RQ 3, we focused on a specific use case, namely the fingerprinting of network traffic from *mobile applications*. We chose this use case as mobile applications are easily installed, removed or updated, changing the characteristics of their network traffic and therefore making it difficult to group traffic per application. Moreover, due to bring-your-own-device policies and the high level of encrypted network traffic, mobile devices are often even less visible to security operators. This makes it even more important to be able to provide contextual information from the network level. We introduced FlowPrint which is built on the assumption that mobile applications consist of various components that often communicate with dedicated network services. By clustering mobile network traffic based on its destination and then correlating these different destinations, we were able to create fingerprints for both known and previously unseen apps. These fingerprints can be provided to security operators as additional information when triaging security events.

With our research questions, we focused on the open problems in the detection and mitigation of security events. However, we also wanted to know if our proposed context-based approaches could help to improve other areas of security. Therefore, we conducted a case study exploring the benefits of context-based anomaly detection in the area of identity and access management (IAM). In this case study, we attempted to detect misconfigurations in IAM policies for AWS cloud environments. Our context-based approach modeled policies as a graph, connecting policies to actions and resources. This graph representation allowed us to embed the context of a policy in terms of relation to the different actions and resources and use this context in combination with anomaly detection to identify possible misconfigurations. By proactively searching for potential misconfigurations we reduced the surface of potential attacks, ultimately leading to fewer security incidents, showing that context-based approaches can benefit other areas of research as well.

In summary, this thesis presented a study that attempted to comprehend security events produced by IDSs and provided context-based approaches to improve the identification and explanation of these events for security operators in a SOC. We attempted to integrate high-level contextual knowledge in three components of the security operator workflow:

1. Threat Intelligence Processing, automating knowledge extraction from high-level CTI reports to enable improved and automated information sharing.

2. DeepCASE, providing contextual analysis of security events by grouping sequences of security events that occur in a similar context semi-automatically.

3. FlowPrint, providing improved contextual information about applications on the network level that can be used during the triaging process.

By doing so, we improved our understanding of security events and used this knowledge to enhance identification and explanation capabilities for security events.

## 6.2 Process Reflection

The observant reader may have noticed that the order of publications on which this thesis is based is different from the order in which works were discussed. In fact, the chronological order in which we worked on various aspects of the thesis started with FlowPrint (2020) [3], then DeepCASE (2022) [1], detecting IAM policy misconfigurations (2022) [2], and lastly threat intelligence processing (2023) [4]. This difference reflects both a change in focus of the PhD project, as well as the development of deeper insights into the process of handling security events.

Initially, this project focused on detecting new and previously unseen security events in the context of the "EVolutionary Intrusion DEtectioN for Changing Environments" (EVIDENCE) project. FlowPrint focused on detecting new events in the context of mobile applications. However, after this project had finished, we found that too little was understood about the subsequent analysis of these events, which was the focus of the DeepCASE chapter. Where DeepCASE proved the importance of contextual analysis, it raised the question of how security operators receive adequate information for making these contextual assessments, leading to our final work on threat intelligence processing.

While this explains the publication order, it does not explain the thesis structure. For the thesis structure, we reflected on the best way to introduce new readers to the processing of security events. Here, we decided to provide a top-down explanation to cement the idea that security events can only be understood in the context of a larger attack. To this end, we started with our work on threat intelligence processing. As the cyber threat intelligence reports that we analyze focus on TTPs, it introduces the reader to the context-aware view that operators have when analyzing security events. The idea is that with this background, the (semi-)automated contextual analysis of security events introduced by DeepCASE is easier to understand. After our attempt to convince the reader that context-based analyses are crucial in the identification and explanation of security events, we could challenge our fundamental underlying assumption that security events provide sufficient information for these analyses. This was the initial problem that we started with in FlowPrint, where simply detecting known applications did not provide enough context to operators, but we also required knowledge of new and previously unseen applications. Therefore, while the work done in FlowPrint was our starting point, we found that it required a deeper understanding of the security events it produced. Hence, our non-chronological thesis structure attempts to provide the reader with a deeper understanding of security events we gained ourselves upfront. We hope that this presents the reader with a more contextual understanding of the various research challenges explored in this work.

## 6.3 Limitations & Future work

While this thesis aims to provide a better understanding of security events through contextual analysis, even after five years of research, we do not consider this a solved problem. To start positively, we have gained a better understanding of extracting kill chain information from CTI; analyzing contextual security events for more efficient triaging; and using network fingerprints to give more insights into active applications. However, as we have seen, in its current form, analysis of security events still requires human intervention to make final decisions about mitigating or discarding potential attacks. Moreover, better solutions to the individual sub-problems that we addressed may exist in terms of performance, computational complexity or explainability, especially with the recent advances in many areas of research that we covered. We do not claim to have completely solved the individual sub-problems, but we did advance the state-of-the-art allowing us to better understand security events. Leaving the individual limitations of our proposed methods to their corresponding chapters, in the remainder of this section, we identify several limitations of the thesis as a whole and propose directions for future work.

### 6.3.1 Environment-based contextualization

The current approaches discussed in this thesis contextualize security events using external threat reports (Chapter 2), other security events (Chapter 3), or additional network data (Chapter 4). However, the severity of a security event, and therefore the triaging process and potential mitigation, also depends on the environment in which the event is observed. To illustrate, a DDoS attack on vital energy infrastructure is considered more severe than a DDoS attack on a personal website. While the attack and its related security events may be the same, the response that security operators propose can vastly differ. For this thesis, the analysis of security events in the context of the environment in which events occur was considered out of scope. However, future work may investigate ways of modeling the monitored IT infrastructure in terms of potential impact for different types of attacks. Such a model could include network-level, device-level or even more fine-grained application-level information on business continuity impact, similar to [78]. Next, such impact assessment can be integrated into triaging tools to improve the timeliness and accuracy of triaging operators.

### 6.3.2 (Semi-)automated kill chain detection

In this thesis, we have considered the human-in-the-loop as a vital step in the triaging of security events. However, security operators, or more specifically security threat analysts also manually create detection rules, heuristics or detectors that trigger the events that operators receive. Just like CTI reports, rules are shared in the form of Yara [222], SIGMA [181], Suricata [85] or other forms of rule-based detection. However, these rules often focus on detecting single malicious events, not capturing larger kill chains. One area of research that does attempt to track various steps of program behavior studies provenance graphs [36]. These provenance graphs can be queried to find malicious behavior on a device [98, 99]. Leveraging threat intelligence from CTI reports, possibly in combination with environment-based contextualization, it may be possible to (semi-)automatically generate detection queries for multi-step kill chains. Advantages of such detection techniques are that they are inherently explainable, as they would be generated from CTI reports describing the attack. However, future research should investigate the detection performance of such rules, and to what extent they can be evaded by APTs.

### 6.3.3 Context-informed response

This thesis has focused on understanding security events with the aim of triaging and ultimately understanding the individual steps of an attack. However, after the security incident has been established, security operators should take steps to mitigate the impact of the attack. To this end, we propose to use information from detected kill chains to automatically propose mitigation steps (known as playbooks) to security operators. Please note that we stress the role of security operators in this step as fully automated intervention leads to both ethical concerns (as users may be incorrectly shut out of their accounts) and practical considerations (do we want an autonomous systems to potentially disrupt vital business processes?). Nevertheless, generating mitigation playbooks could potentially save security operators a lot of work, especially when combined with environment-based contextual knowledge (see subsection 6.3.1) about the systems under attack.

# Acknowledgements

When I began my studies at the University of Twente in 2012, I had never heard of the concept of a PhD nor had any intention of pursuing one. It was only during my master thesis that I found out I enjoyed doing research, which led me to start my PhD in 2018. Now, five years of PhD experience later, I have learned that a PhD trajectory does not just involve research, but helps you grow as a person. The experiences I have gained over the past few years have been invaluable, and I cannot imagine how my life would have turned out if I had made different decisions. While experience is merely the name that people give to their mistakes, I would like to thank the all people who helped me through my experiences.

I would like to thank my team of supervisors: Andreas, Maarten and Andrea. Andreas, thank you for supporting my decision to pursue a PhD, for allowing me to make mistakes such that I could grow, and most of all, thank you for the sharp questions during my progress meetings that were vital in the formulation of my research. Maarten, thank you for your kindness and positivity, your amazing capacity for understanding my technical gibberish and your incessant encouragement to help me see the bigger picture. Andrea, thank you for your enthusiasm for my research ideas, even the crazy ones, for welcoming me into what feels like a research family, and for your continued guidance and support during my academic career.

Thank you to the committee members for taking the time to read my work and provide valuable feedback.

Riccardo, I would like to especially thank you for working with me. Without you, I would have never started this PhD adventure nor would I have had so many fun evenings playing board games with you, Roeland and Tim. Similarly, I would also like to thank all other fellow PhD and EngD students who made this ride so enjoyable. Those who came before me and welcomed me into a fun and warm environment, thank you Ali, Bence, Chris, Dan, Erwin, Herson, Philipp, Riccardo, Roeland, Susanne, Tim and Valeriu. To those who started their journey after me, thank you for continuing to make this group a joy to work in: Amina, Asbat, Chakshu, Claudio, Donika, Faezeh, Farideh, Federico, Filipi, Hudi, Isadora, Ítalo, Jacco, Jerre, Jorik, Kemilly, Marc, Matteo, Meikel, Pedro, Reza, Rodrigo, Taru, Una, Yikun, Yoep, and Zsolt.

Next, I would like to thank Bertine, Geert Jan, Jeanette and Suse for all your support, without you our group would be lost. As well as the other previous and current staff members of the SCS group, Andrea, Andreas, Claudenir, Dipti, Erik, Giancarlo, Florian, Jeewanie, João, Leon, Luca, Luis, Luiz, Maarten, Marten, Maya, Mohammed, Pieter, Roel, Thijs, Tiago, Tom, Wallace and Willem.

Besides the SCS research group, I had the privilege to work together with some of the most wonderful researchers I could imagine. I would like to thank Chris and Giovanni, for their guidance and collaboration, allowing me to join their magnificent research group. Here, I met new colleagues and friends, Andrea, Chad, Colin, Dipanjan, Eric, Fabian, both Fabio's (French and Italian), Francesco, Hector, Hojjat, Kevin, Lucas, Machiry, Nilo, Noah, Priyanka, Robert, Saastha, Salls, and Seba. Also thank you to my other co-authors Dan, Daniel, David, Gal, Hodaya, Jingjing, Marco Caselli, Marco Cova and Martina for your support and interesting discussions that helped shape our research.

During my PhD, I was also part of the emergency response team of the Waaier. Therefore, I would like to thank Alfred, Bernd, Daniel, Diana, Elise, Henk, Henk-Willem, Geert Jan, Jan, Johan, Josca, Marthe, Martijn, Mattijs, Michel, Miranda, Ömer, Raymond, Victor, Yakup and Yoep for their help during training and emergencies.

Next to work, I have an awesome group of friends and family who were supportive during the entire PhD and allowed me to free my mind of work-related things. Bjorn and Thomas, you probably know me better than I know myself, therefore I find it an honor that you could stand by my side as my paranymphs. Thank you for being there throughout all ups and downs, it really means a lot to me. Arthur, Luka and Ruben, thank you for all the fun days and nights we spent together, for sticking with me and still laughing at my lame jokes, even after 15 years of friendship. Anirudh and Sem, thank you for showing me that security and fun go hand in hand. Thank you to my pub quiz team, Bjorn, Chris, Erwin, Janina, Jelle, Jerre, Jorik, Jeroen, Lena, Philipp, Rutger, Steyn, Thomas and Yoep for, in the words of quizmaster Tom, "all the completely useless nights of gezelligheid".

Finally, I would like to thank my family. Mama, papa en Joris, dank jullie wel dat jullie er altijd voor mij zijn geweest en mij gesteund hebben in alles wat ik deed. Dank jullie wel voor het meegeven van een basis van vertrouwen, jullie onvoorwaardelijke steun om op terug te vallen, en de instelling om alle problemen die ik tegenkom bij de horens te grijpen. Zonder jullie had ik dit alles nooit gekund.

I would like to end my acknowledgements with a quote that, in my mind, sums up the PhD experience:

> *"Success is not final, failure is not fatal: it is the courage to continue that counts."* (Winston S. Churchill)

Thijs Sebastiaan van Ede
Enschede, November 2, 2023

# Bibliography

## Author's Publications

[1] Thijs van Ede, Hojjat Aghakhani, Noah Spahn, Riccardo Bortolameotti, Marco Cova, Andrea Continella, Maarten van Steen, Andreas Peter, Christopher Kruegel, and Giovanni Vigna. “DeepCASE: Semi-supervised contextual analysis of security events”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 522–539.

[2] Thijs van Ede, Niek Khasuntsev, Bas Steen, and Andrea Continella. “Detecting Anomalous Misconfigurations in AWS Identity and Access Management Policies”. In: *Proceedings of the 2022 on Cloud Computing Security Workshop (CCSW)*. 2022, pp. 63–74.

[3] Thijs van Ede, Riccardo Bortolameotti, Andrea Continella, Jingjing Ren, Daniel J Dubois, Martina Lindorfer, David Choffnes, Maarten van Steen, and Andreas Peter. “Flowprint: Semi-supervised mobile-app fingerprinting on encrypted network traffic”. In: *Network and Distributed System Security Symposium (NDSS)*. Vol. 27. 2020.

[4] Thijs van Ede, Hodaya Binyamini, Gal Engelberg, Dan Klein, Giancarlo Guizzardi, Marco Caselli, Andrea Continella, Maarten van Steen, and Andreas Peter. “Threat Intelligence Processing - Unleashing the Real Power of Natural Language Processing for Cyber Threat Intelligence”. In: *(Under submission)*. 2023.

[5] Riccardo Bortolameotti, Thijs van Ede, Marco Caselli, Maarten H Everts, Pieter Hartel, Rick Hofstede, Willem Jonker, and Andreas Peter. “DECANTeR: DEteCtion of Anomalous outbouNd HTTP TRaffic by Passive Application Fingerprinting”. In: *Proc. of the Annual Computer Security Applications Conference (ACSAC)*. 2017.

[6] Max Meijer, Giacomo Tommaso Petrucci, Matthijs Schotsman, Luca Morgese Zangrandi, Thijs van Ede, Andrea Continella, Ganduulga Gankhuyag, Luca Allodi, and Savio Sciancalepore. “Federated Lab (FedLab): An Open-source Distributed Platform for Internet of Things (IoT) Research and Experimentation”. In: *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*. IEEE. 2022, pp. 1–8.

[7] Riccardo Bortolameotti, Thijs van Ede, Andrea Continella, Thomas Hupperich, Maarten Everts, Reza Rafati, Willem Jonker, Pieter Hartel, and Andreas Peter. "HeadPrint: Detecting Anomalous Communications through Header-based Application Fingerprinting". In: *Proc. of the ACM Symposium on Applied Computing (SAC)*. 2020.

[8] Chakshu Gupta, Thijs van Ede, and Andrea Continella. "HoneyKube: Designing and Deploying a Microservices-based Web Honeypot". In: *Workshop on Designing Security for the Web (SecWeb)*. 2023.

[9] Fatemeh Marzani, Fatemeh Ghassemi Esfahani, Zeynab Sabahi-Kaviani, Thijs van Ede, and Maarten van Steen. "Mobile App Fingerprinting through Automata Learning and Machine Learning". In: *Proceedings of the 22nd International Federation for Information Processing Networking Conference (IFIP Networking)*. 2023.

[10] Luca Morgese Zangrandi, Thijs van Ede, Tim Booij, Savio Sciancalepore, Luca Allodi, and Andrea Continella. "Stepping out of the MUD: Contextual threat information for IoT devices with manufacturer-provided behavior profiles". In: *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC)*. 2022, pp. 467–480.

[11] Riccardo Bortolameotti, Thijs van Ede, Andrea Continella, Maarten Everts, Willem Jonker, Pieter Hartel, and Andreas Peter. "Victim-Aware Adaptive Covert Channels". In: *Security and Privacy in Communication Networks: 15th EAI International Conference (SecureComm)*. Springer. 2019, pp. 450–471.

## Literature

[12] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and A. Selcuk Uluagac. "Peek-a-Boo: I see your smart home activities, even encrypted!" In: *arXiv preprint arXiv:1808.02741* (2018).

[13] Giuseppe Aceto, Domenico Ciuonzo, Antonio Montieri, and Antonio Pescapé. "Multi-Classification Approaches for Classifying Mobile App Traffic". In: *Journal of Network and Computer Applications* (2018).

[14] Jagdish Prasad Achara, Gergely Acs, and Claude Castelluccia. "On the Unicity of Smartphone Applications". In: *Proc. of the ACM Workshop on Privacy in the Electronic Society (WPES)*. 2015.

[15] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. "When Malware is Packin'Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features". In: *Network and Distributed Systems Security Symposium (NDSS)*. 2020.

[16] Hasan Faik Alan and Jasleen Kaur. "Can Android Applications Be Identified Using Only TCP/IP Headers of Their Launch Time Traffic?" In: *Proc. of the ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*. 2016.

[17] John B. Althouse, Jeff Atkinson, and Josh Atkins. *JA3 - A method for profiling SSL/TLS Clients*. `https://github.com/salesforce/ja3`. June 2017.

[18] Paulo MMR Alves, PR Geraldo Filho, and Vinícius P Gonçalves. "Leveraging BERT's Power to Classify TTP from Unstructured Text". In: *2022 Workshop on Communication Networks and Power Systems (WCNPS)*. IEEE. 2022, pp. 1–7.

[19] AmazonWebServices. *AWS IAM*. 2003. URL: `https://aws.amazon.com/iam/`.

[20] Saleema Amershi, Bongshin Lee, Ashish Kapoor, Ratul Mahajan, and Blaine Christian. "Human-guided machine learning for fast and accurate network alarm triage". In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI. 2016.

[21] Muhamad Erza Aminanto, Lei Zhu, Tao Ban, Ryoichi Isawa, Takeshi Takahashi, and Daisuke Inoue. "Combating Threat-Alert Fatigue with Online Anomaly Detection Using Isolation Forest". In: *International Conference on Neural Information Processing*. Springer. 2019, pp. 756–765.

[22] Blake Anderson and David McGrew. "Identifying Encrypted Malware Traffic with Contextual Flow Data". In: *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*. 2016.

[23] Blake Anderson and David McGrew. "TLS Beyond the Browser: Combining End Host and Network Data to Understand Application Behavior". In: *Proc. of the ACM Internet Measurement Conference (IMC)*. 2019.

[24] Blake Anderson, Subharthi Paul, and David McGrew. "Deciphering Malware's use of TLS (without Decryption)". In: *Journal of Computer Virology and Hacking Techniques* (2018).

[25] AppAnnie. *The State of Mobile in 2019*. `https://www.appannie.com/de/insights/market-data/the-state-of-mobile-2019/`. Jan. 2019.

[26] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. “Dos and don’ts of machine learning in computer security”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 3971–3988.

[27] Vani Asawa. *Extract Actionable Intelligence from Text-based Threat Intel using Sentinel Notebook*. `https://techcommunity.microsoft.com/t5/microsoft-sentinel-blog/what-s-new-extract-actionable-intelligence-from-text-based/ba-p/3729508`, retrieved 2023-04-11. 2023.

[28] AWS. *Amazon Simple Storage Service (Amazon S3)*. 2002. URL: `https://aws.amazon.com/s3/`.

[29] AWS. *AWS IAM Best Practices*. 2003. URL: `https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html`.

[30] *AWS CloudTrail*. 2021. URL: `https://aws.amazon.com/cloudtrail/`.

[31] *AWS Command Line Interface*. 2006. URL: `https://aws.amazon.com/cli/`.

[32] Michael Backes, Sven Bugiel, and Erik Derr. “Reliable Third-Party Library Detection in Android and its Security Applications”. In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2016.

[33] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).

[34] Taimur Bakhshi and Bogdan Ghita. “On Internet Traffic Classification: A Two-Phased Machine Learning Approach”. In: *Journal of Computer Networks and Communications* (2016).

[35] Vimala Balakrishnan and Ethel Lloyd-Yemoh. “Stemming and lemmatization: A comparison of retrieval performances”. In: (2014).

[36] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. “Trustworthy Whole-System provenance for the linux kernel”. In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 319–334.

[37] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. “A neural probabilistic language model”. In: *Advances in Neural Information Processing Systems* 13 (2000).

[38] Kelly W Bennett and James Robertson. “Security in the Cloud: understanding your responsibility”. In: *Cyber Sensing 2019*. Vol. 11011. International Society for Optics and Photonics. 2019, p. 1101106.

[39] Jon Louis Bentley. “Multidimensional binary search trees used for associative searching”. In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[40] Laurent Bernaille, Renata Teixeira, Ismael Akodkenou, Augustin Soule, and Kave Salamatian. "Traffic Classification On The Fly". In: *ACM SIGCOMM Computer Communication Review* (2006).

[41] Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. "Falling Asleep with Angry Birds, Facebook and Kindle – A Large Scale Study on Mobile Application Usage". In: *Proc. of the International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI).* 2011.

[42] Bernd Bohnet, Chris Alberti, and Michael Collins. "Coreference Resolution through a seq2seq Transition-Based System". In: *arXiv preprint arXiv:2211.12142* (2022).

[43] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. "LOF: identifying density-based local outliers". In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data.* 2000, pp. 93–104.

[44] JM Brook, A Getsin, G Jensen, L Jameson, M Roza, N Thethi, A Kurmi, S Levy, S Shamban, V Hargrave, et al. "Top Threats to Cloud Computing: The Egregious Eleven". In: *Cloud Security Alliance* (2019).

[45] Andy Brown, Aaron Tuor, Brian Hutchinson, and Nicole Nichols. "Recurrent neural network attention mechanisms for interpretable system log anomaly detection". In: *Proceedings of the First Workshop on Machine Learning for Computing Systems.* 2018, pp. 1–8.

[46] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[47] Herbert G. Buff. *Compliance.* 2000. URL: `https://aws.amazon.com/compliance/shared-responsibility-model/`.

[48] Juan Caballero, Gibran Gomez, Srdjan Matic, Gustavo Sánchez, Silvia Sebastián, and Arturo Villacañas. "GoodFATR: A Platform for Automated Threat Report Collection and IOC Extraction". In: *arXiv preprint arXiv:2208.00042* (2022).

[49] Raghavendra Chalapathy, Aditya Krishna Menon, and Sanjay Chawla. "Anomaly detection using one-class neural networks". In: *arXiv preprint arXiv:1802.06360* (2018).

[50] Yi-Chao Chen, Yong Liao, Mario Baldi, Sung-Ju Lee, and Lili Qiu. "OS Fingerprinting and Tethering Detection in Mobile Networks". In: *Proc. of the ACM Internet Measurement Conference (IMC).* 2014.

[51] Danqi Chen and Christopher D Manning. “A fast and accurate dependency parser using neural networks”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 740–750.

[52] Yi Chen, Wei You, Yeonjoon Lee, Kai Chen, XiaoFeng Wang, and Wei Zou. “Mass Discovery of Android Traffic Imprints through Instantiated Partial Execution”. In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2017.

[53] Zhengyang Chen, Bowen Yu, Yu Zhang, Jianzhong Zhang, and Jingdong Xu. “Automatic Mobile Application Traffic Identification by Convolutional Neural Networks”. In: *Proc. of IEEE Trustcom/BigDataSE/ISPA*. 2016.

[54] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078* (2014).

[55] Yeongrak Choi, Jae Yoon Chung, Byungchul Park, and James Won-Ki Hong. “Automated Classifier Generation for Application- Level Mobile Traffic Identification”. In: *Proc. of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*. 2012.

[56] Tobias Chyssler, Stefan Burschka, Michael Semling, Tomas Lingvall, and Kalle Burbeck. “Alarm reduction and correlation in intrusion detection systems”. In: *Detection of intrusions and malware & vulnerability assessment, GI SIG SIDAR workshop, DIMVA 2004*. Gesellschaft für Informatik eV. 2004.

[57] Casey Cipriano, Ali Zand, Amir Houmansadr, Christopher Kruegel, and Giovanni Vigna. “Nexat: A history-based approach to predict attacker actions”. In: *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*. 2011, pp. 383–392.

[58] CISCO. *Anticipating the Unknowns - Chief Information Security Officer (CISO) Benchmark Study*. Tech. rep. Retrieved from `https://ebooks.cisco.com/story/anticipating-unknowns/` on 2020-9-16. 2019.

[59] Cisco. *SNORT Users Manual*. `https://www.snort.org/documents/snort-users-manual`. Cisco, 2018.

[60] Stefano Comino, Fabio M. Manenti, and Franco Mariuzzo. “Updates Management in Mobile Applications. iTunes vs Google Play”. In: *Journal of Economics & Management Strategy* (2018).

[61] Mauro Conti, Luigi V. Mancini, Riccardo Spolaor, and Nino Vincenzo Verde. “Can’t You Hear Me Knocking: Identification of User Actions on Android Apps via Traffic Analysis”. In: *Proc. of the ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2015.

[62] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. "Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis". In: *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS).* 2017.

[63] Andrea Continella, Mario Polino, Marcello Pogliani, and Stefano Zanero. "There's a Hole in that Bucket! A Large-scale Analysis of Misconfigured S3 Buckets". In: *Proceedings of the ACM Annual Computer Security Applications Conference (ACSAC).* Dec. 2018.

[64] Frédéric Cuppens and Alexandre Miege. "Alert correlation in a cooperative intrusion detection framework". In: *Proceedings 2002 IEEE symposium on security and privacy.* IEEE. 2002, pp. 202–215.

[65] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. "NetworkProfiler: Towards Automatic Fingerprinting of Android Apps". In: *Proc. of the IEEE International Conference on Computer Communications (INFOCOM).* 2013.

[66] Tathagata Das, Ranjita Bhagwan, and Prasad Naldurg. "Baaz: A System for Detecting Access Control Misconfigurations." In: *USENIX Security Symposium.* 2010, pp. 161–176.

[67] Marie-Catherine De Marneffe, Timothy Dozat, Natalia Silveira, Katri Haverinen, Filip Ginter, Joakim Nivre, and Christopher D Manning. "Universal Stanford dependencies: A cross-linguistic typology." In: *LREC.* Vol. 14. 2014, pp. 4585–4592.

[68] Marie-Catherine De Marneffe, Bill MacCartney, Christopher D Manning, et al. "Generating typed dependency parses from phrase structure parses." In: *Lrec.* Vol. 6. 2006, pp. 449–454.

[69] Demisto. *The State of SOAR Report.* Tech. rep. Retrieved from `https://start.paloaltonetworks.com/the-state-of-soar-report-2018` on 2020-9-16. 2018.

[70] Universal Dependencies. *Universal Dependency Relations.* `https://universaldependencies.org/u/dep/`, retrieved 2023-04-12. 2014.

[71] Universal Dependencies. *Universal POS tags.* `https://universaldependencies.org/u/pos/`, retrieved 2023-04-12. 2014.

[72] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[73] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. "Investigating system operators' perspective on security misconfigurations". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 1272–1289.

[74] Trinh Minh Tri Do and Daniel Gatica-Perez. "Where and What: Using Smartphones to Predict Next Locations and Applications in Daily Life". In: *Pervasive and Mobile Computing* (2014).

[75] Min Du, Zhi Chen, Chang Liu, Rajvardhan Oak, and Dawn Song. "Lifelong Anomaly Detection Through Unlearning". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2019, 1283–1297.

[76] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. "Deeplog: Anomaly detection and diagnosis from system logs through deep learning". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017, 1285–1298.

[77] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J Alex Halderman, and Vern Paxson. "The Security Impact of HTTPS Interception". In: *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*. 2017.

[78] Herson Esquivel-Vargas, Marco Caselli, Erik Tews, Doina Bucur, and Andreas Peter. "BACRank: ranking building automation and control system components by business continuity impact". In: *Computer Safety, Reliability, and Security: 38th International Conference, SAFECOMP 2019, Turku, Finland, September 11–13, 2019, Proceedings 38*. Springer. 2019, pp. 183–199.

[79] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. Vol. 96. 34. 1996, pp. 226–231.

[80] Shuhan Fan, Songyun Wu, Zhiliang Wang, Zimu Li, Jiahai Yang, Heng Liu, and Xinran Liu. "ALEAP: Attention-based LSTM with Event Embedding for Attack Projection". In: *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*. IEEE. 2019, pp. 1–8.

[81] Flatironhealth. *Aws Remediation Framework*. 2021. URL: `https://github.com/flatironhealth/aws-remediation-framework`.

[82] Emily Flitter and Karen Weise. *Capital One Data Breach Compromises Data of Over 100 Million*. July 2019. URL: `https://www.nytimes.com/2019/07/29/business/capital-one-data-breach-hacked.html`.

[83] David Formby, Preethi Srinivasan, Andrew Leonard, Jonathan Rogers, and Raheem Beyah. "Who's in Control of Your Control System? Device Fingerprinting for Cyber-Physical Systems". In: *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS).* 2016.

[84] Linux Foundation. *Cloud Custodian.* 2020. URL: `https://cloudcustodian.io/`.

[85] Open Information Security Foundation. *Suricata.* 2009. URL: `https://suricata.io/`.

[86] Frolov, Sergey and Wustrow, Eric. "The use of TLS in Censorship Circumvention". In: *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS).* 2019.

[87] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. "An Analysis of Pre-installed Android Software". In: *Proc. of the IEEE Symposium on Security and Privacy (S&P).* 2020.

[88] Peng Gao, Xiaoyuan Liu, Edward Choi, Sibo Ma, Xinyu Yang, Zhengjie Ji, Zilin Zhang, and Dawn Song. "ThreatKG: A Threat Knowledge Graph for Automated Open-Source Cyber Threat Intelligence Gathering and Management". In: *arXiv preprint arXiv:2212.10388* (2022).

[89] Peng Gao, Fei Shao, Xiaoyuan Liu, Xusheng Xiao, Zheng Qin, Fengyuan Xu, Prateek Mittal, Sanjeev R Kulkarni, and Dawn Song. "Enabling efficient cyber threat hunting with cyber threat intelligence". In: *2021 IEEE 37th International Conference on Data Engineering (ICDE).* IEEE. 2021, pp. 193–204.

[90] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics.* 2011, pp. 315–323.

[91] Google. *An Update on Android TLS Adoption.* `https://security.googleblog.com/2019/12/an-update-on-android-tls-adoption.html`. Dec. 2019.

[92] Cyril Goutte and Eric Gaussier. "A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation". In: *Proc. of the European Conference on Information Retrieval (ECIR).* 2005.

[93] Palash Goyal and Emilio Ferrara. "Graph embedding techniques, applications, and performance: A survey". In: *Knowledge-Based Systems* 151 (2018), pp. 78–94.

[94] *Graph Database Platform: Graph Database Management System: Neo4j.* May 2021. URL: `https://neo4j.com/`.

[95] Aditya Grover and Jure Leskovec. "node2vec: Scalable feature learning for networks". In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016, pp. 855–864.

[96] Aric Hagberg, Daniel Schult, and Pieter Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proc. of the Python in Science Conference (SciPy)*. 2008.

[97] William L Hamilton, Rex Ying, and Jure Leskovec. "Inductive representation learning on large graphs". In: *arXiv preprint arXiv:1706.02216* (2017).

[98] Xueyuan Han, Thomas F. J.-M. Pasquier, Adam Bates, James Mickens, and Margo I. Seltzer. "Unicorn: Runtime Provenance-Based Detector for Advanced Persistent Threats". In: *27th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2020. URL: `https://www.ndss-symposium.org/ndss-paper/unicorn-runtime-provenance-based-detector-for-advanced-persistent-threats/`.

[99] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. "Nodoze: Combatting threat alert fatigue with automated provenance triage". In: *network and distributed systems security symposium*. 2019.

[100] Wajih Ul Hassan, Mohammad A Noureddine, Pubali Datta, and Adam Bates. "Omega-Log: High-fidelity attack investigation via transparent multi-layer log analysis". In: *Network and Distributed Systems Security Symposium (NDSS)*. 2020.

[101] Grant Ho, Aashish Sharma, Mobin Javed, Vern Paxson, and David Wagner. "Detecting credential spearphishing in enterprise settings". In: *26th USENIX Security Symposium*. 2017, pp. 469–485.

[102] Tin Kam Ho. "Random Decision Forests". In: *Proc. of IEEE International Conference on Document Analysis and Recognition (ICDAR)*. 1995.

[103] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[104] Matthew Honnibal and Mark Johnson. "An Improved Non-monotonic Transition System for Dependency Parsing". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sept. 2015, pp. 1373–1378. DOI: `10.18653/v1/D15-1162`. URL: `https://aclanthology.org/D15-1162`.

[105] Xin Hu, Jiyong Jang, Marc Ph Stoecklin, Ting Wang, Douglas L Schales, Dhilung Kirat, and Josyula R Rao. “BAYWATCH: robust beaconing detection to identify infected hosts in large-scale enterprise networks”. In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2016, pp. 479–490.

[106] Richard Hudson. *Coreferee*. `https://github.com/richardpaulhudson/coreferee`, retrieved 2023-04-12. 2022.

[107] Ghaith Husari, Ehab Al-Shaer, Mohiuddin Ahmed, Bill Chu, and Xi Niu. “Ttpdrill: Automatic and accurate extraction of threat actions from unstructured text of cti sources”. In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. 2017, pp. 103–115.

[108] Paul Jaccard. “The Distribution of the Flora of the Alpine Zone”. In: *New Phytologist* (1912).

[109] jackaduma. *SecBERT*. `https://github.com/jackaduma/SecBERT`, retrieved 2023-04-12. 2020.

[110] Sarthak Jain and Byron C Wallace. “Attention is not explanation”. In: *arXiv preprint arXiv:1902.10186* (2019).

[111] Chris Johnson, Lee Badger, David Waltermire, Julie Snyder, and Clem Skorupka. *Guide to Cyber Threat Information Sharing*. Tech. rep. NIST, 2016. DOI: `http://dx.doi.org/10.6028/NIST.SP.800-150`.

[112] Mandar Joshi, Omer Levy, Daniel S Weld, and Luke Zettlemoyer. “BERT for coreference resolution: Baselines and analysis”. In: *arXiv preprint arXiv:1908.09091* (2019).

[113] Peter E. Kaloroumakis and Michael J. Smith. *Toward a Knowledge Graph of Cybersecurity Countermeasures*. Tech. rep. `https://d3fend.mitre.org/`, retrieved 2023-04-12. The MITRE Corporation, 2021.

[114] Muhammad Kazim and Shao Ying Zhu. “A survey on top security threats in cloud computing”. In: (2015).

[115] Issa M Khalil, Abdallah Khreishah, Salah Bouktif, and Azeem Ahmad. “Security concerns in cloud computing”. In: *2013 10th International Conference on Information Technology: New Generations*. IEEE. 2013, pp. 411–416.

[116] Thomas N Kipf and Max Welling. “Semi-supervised classification with graph convolutional networks”. In: *arXiv preprint arXiv:1609.02907* (2016).

[117] Dan Klein and Christopher D Manning. “Fast Exact Inference with a Factored Model for Natural Language Parsing”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Becker, S. Thrun, and K. Obermayer. Vol. 15. MIT Press, 2002. URL: `https://proceedings.neurips.cc/paper/2002/file/6c97cd07663b099253bc569fe8d342bb-Paper.pdf`.

[118] Faris Bugra Kokulu, Ananta Soneji, Tiffany Bao, Yan Shoshitaishvili, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. “Matched and mismatched SOCs: A qualitative study on security operations center issues”. In: *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2019, pp. 1955–1970.

[119] Sandra Kübler, Ryan McDonald, and Joakim Nivre. “Dependency parsing”. In: *Synthesis lectures on human language technologies* 1.1 (2009), pp. 1–127.

[120] Solomon Kullback and Richard A Leibler. “On information and sufficiency”. In: *The annals of mathematical statistics* 22.1 (1951), pp. 79–86.

[121] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. “Neural architectures for named entity recognition”. In: *arXiv preprint arXiv:1603.01360* (2016).

[122] Martin Lastovicka, Tomas Jirsik, Pavel Celeda, Stanislav Spacek, and Daniel Filakovsky. “Passive OS Fingerprinting Methods in the Jungle of Wireless Networks”. In: *Proc. of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*. 2018.

[123] Kenton Lee, Luheng He, Mike Lewis, and Luke Zettlemoyer. “End-to-end neural coreference resolution”. In: *arXiv preprint arXiv:1707.07045* (2017).

[124] Valentine Legoy, Marco Caselli, Christin Seifert, and Andreas Peter. “Automated retrieval of att&ck tactics and techniques for cyber threat reports”. In: *arXiv preprint arXiv:2004.14322* (2020).

[125] Fangfan Li, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. “A Large-Scale Analysis of Deployed Traffic Differentiation Practices”. In: *Proc. of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 2019.

[126] Jing Li, Aixin Sun, Jianglei Han, and Chenliang Li. “A survey on deep learning for named entity recognition”. In: *IEEE Transactions on Knowledge and Data Engineering* 34.1 (2020), pp. 50–70.

[127] Li Li, Daoyuan Li, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. “Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting”. In: *IEEE Transactions on Information Forensics and Security* (2017).

[128] Zhenyuan Li, Jun Zeng, Yan Chen, and Zhenkai Liang. "AttacKG: Constructing Technique Knowledge Graph from Cyber Threat Intelligence Reports". In: *arXiv preprint arXiv:2111.07093* (2021).

[129] Zuchao Li, Jiaxun Cai, Shexia He, and Hai Zhao. "Seq2seq dependency parsing". In: *Proceedings of the 27th International Conference on Computational Linguistics.* 2018, pp. 3203–3214.

[130] Xiaojing Liao, Kan Yuan, XiaoFeng Wang, Zhou Li, Luyi Xing, and Raheem Beyah. "Acing the ioc game: Toward automatic discovery and analysis of open-source cyber threat intelligence". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security.* 2016, pp. 755–766.

[131] Marc Liberatore and Brian Neil Levine. "Inferring the Source of Encrypted HTTP Connections". In: *Proc. of the ACM Conference on Computer and Communications Security (CCS).* 2006.

[132] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. "Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors". In: *Proc. of the IEEE International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS).* 2014.

[133] Fucheng Liu, Yu Wen, Dongxue Zhang, Xihe Jiang, Xinyu Xing, and Dan Meng. "Log2vec: A Heterogeneous Graph Embedding Based Approach for Detecting Cyber Threats within Enterprise". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS).* 2019, pp. 1777–1794.

[134] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. "Roberta: A robustly optimized bert pretraining approach". In: *arXiv preprint arXiv:1907.11692* (2019).

[135] Christopher D Manning. "Part-of-speech tagging from 97% to 100%: is it time for some linguistics?" In: *International conference on intelligent text processing and computational linguistics.* Springer. 2011, pp. 171–189.

[136] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. "The Stanford CoreNLP natural language processing toolkit". In: *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations.* 2014, pp. 55–60.

[137] David McGrew, Blake Anderson, Philip Perricone, and Bill Hudson. *JOY.* `https://github.com/cisco/joy/`. Feb. 2017.

[138] David McGrew, Brandon Enright, Blake Anderson, Shekhar Acharya, and Adam Weller. *Mercury.* `https://github.com/cisco/mercury`. Aug. 2019.

[139] Wes McKinney et al. "Data Structures for Statistical Computing in Python". In: *Proc. of the Python in Science Conference (SciPy).* 2010.

[140] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs." In: *IJCAI.* Vol. 19. 7. 2019, pp. 4739–4745.

[141] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[142] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, Ramachandran Sekar, and VN Venkatakrishnan. "Holmes: real-time apt detection through correlation of suspicious information flows". In: *2019 IEEE Symposium on Security and Privacy (S&P).* IEEE. 2019, pp. 1137–1152.

[143] George A Miller. *WordNet: An electronic lexical database.* MIT press, 1998.

[144] *Misconfigured AWS S3 Bucket Leaks 36,000 Inmate Records.* Feb. 2020. URL: `https://www.trendmicro.com/vinfo/us/security/news/virtualization-and-cloud/misconfigured-aws-s3-bucket-leaks-36-000-inmate-records`.

[145] Stanislav Miskovic, Gene Moo Lee, Yong Liao, and Mario Baldi. "AppPrint: Automatic Fingerprinting of Mobile Applications in Network Traffic". In: *Proc. of the International Conference on Passive and Active Network Measurement (PAM).* 2015.

[146] MITRE. *Common Attack Pattern Enumeration and Classification.* `https://capec.mitre.org/`, retrieved 2023-04-12. 2007.

[147] MITRE. *Common Vulnerabilities and Exposures.* `https://cve.mitre.org/`, retrieved 2023-04-12. 1999.

[148] MITRE. *Common Weakness Enumeration.* `https://cwe.mitre.org/`, retrieved 2023-04-12. 2006.

[149] Khaled Al-Naami, Swarup Chandra, Ahmad Mustafa, Latifur Khan, Zhiqiang Lin, Kevin Hamlen, and Bhavani Thuraisingham. "Adaptive Encrypted Traffic Fingerprinting With Bi-Directional Dependence". In: *Proc. of the Annual Computer Security Applications Conference (ACSAC).* 2016.

[150] Nag. *S3-Leaks.* 2021. URL: `https://github.com/nagwww/s3-leaks`.

[151] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. "graph2vec: Learning distributed representations of graphs". In: *arXiv preprint arXiv:1707.05005* (2017).

[152] Amirreza Niakanlahiji, Jinpeng Wei, and Bei-Tseng Chu. "A natural language processing based trend analysis of advanced persistent threat techniques". In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 2995–3000.

[153] Joakim Nivre, Marie-Catherine De Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, et al. "Universal dependencies v1: A multilingual treebank collection". In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*. 2016, pp. 1659–1666.

[154] Joakim Nivre and Jens Nilsson. "Pseudo-Projective Dependency Parsing". In: *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*. Ann Arbor, Michigan: Association for Computational Linguistics, June 2005, pp. 99–106. DOI: 10.3115/1219840.1219853. URL: https://aclanthology.org/P05-1013.

[155] OASIS Open. *STIX Version 2.1*. Tech. rep. https://oasis-open.github.io/cti-documentation/, retrieved 2023-04-12. OASIS, 2021.

[156] Travis E. Oliphant. *A Guide to NumPy*. Trelgol Publishing, 2006.

[157] Salima Omar, Asri Ngadi, and Hamid H Jebur. "Machine learning techniques for anomaly detection: an overview". In: *International Journal of Computer Applications* 79.2 (2013).

[158] OpenAI. *Introducing ChatGPT*. https://openai.com/blog/chatgpt/, retrieved 2023-04-12. 2023.

[159] Sinno Jialin Pan and Qiang Yang. "A survey on transfer learning". In: *IEEE Transactions on knowledge and data engineering* 22.10 (2009), pp. 1345–1359.

[160] Ioannis Papapanagiotou, Erich M Nahum, and Vasileios Pappas. "Configuring DHCP Leases in the Smartphone Era". In: *Proc. of the ACM Internet Measurement Conference (IMC)*. 2012.

[161] Vern Paxson. "Bro: A system for detecting network intruders in real-time". In: *Computer networks* 31.23-24 (1999), pp. 2435–2463.

[162] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[163] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. “TESSERACT: Eliminating experimental bias in malware classification across space and time”. In: *28th USENIX Security Symposium.* 2019, pp. 729–746.

[164] Jeffrey Pennington, Richard Socher, and Christopher D Manning. “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP).* 2014, pp. 1532–1543.

[165] Emanuele Petagna, Giuseppe Laurenza, Claudio Ciccotelli, and Leonardo Querzoni. “Peel the Onion: Recognition of Android Apps Behind the Tor Network”. In: *Proc. of the International Conference on Information Security Practice and Experience (ISPEC).* 2019.

[166] Joël Plisson, Nada Lavrac, Dunja Mladenic, et al. “A rule based approach to word lemmatization”. In: *Proceedings of IS.* Vol. 3. 2004, pp. 83–86.

[167] MISP project. *Openn Source Threat Intelligence and Sharing Platform.* `https://www.misp-project.org/`, retrieved 2023-04-12. 2023.

[168] Moumita Das Purba, Bill Chu, and Ehab Al-Shaer. “From Word Embedding to Cyber-Phrase Embedding: Comparison of Processing Cybersecurity Texts”. In: *2020 IEEE International Conference on Intelligence and Security Informatics (ISI).* IEEE. 2020, pp. 1–6.

[169] Lawrence R. Rabiner and Bernard Gold. *Theory and Application of Digital Signal Processing*. Prentice Hall, 1975.

[170] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. “Improving language understanding by generative pre-training”. In: (2018).

[171] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. “Squad: 100,000+ questions for machine comprehension of text”. In: *arXiv preprint arXiv:1606.05250* (2016).

[172] Priyanka Ranade, Aritran Piplai, Anupam Joshi, and Tim Finin. “CyBERT: Contextualized Embeddings for the Cybersecurity Domain”. In: *2021 IEEE International Conference on Big Data (Big Data).* IEEE. 2021, pp. 3334–3342.

[173] Priyanka Ranade, Aritran Piplai, Sudip Mittal, Anupam Joshi, and Tim Finin. “Generating fake cyber threat intelligence using transformer-based models”. In: *2021 International Joint Conference on Neural Networks (IJCNN).* IEEE. 2021, pp. 1–9.

[174] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. “Studying TLS Usage in Android Apps”. In: *Proc. of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT).* 2017.

[175] *Related Words*. `https://relatedwords.org/`, retrieved 2023-04-12. 2023.

[176] Jingjing Ren, Daniel J. Dubois, and David Choffnes. *An International View of Privacy Risks for Mobile Apps*. 2019.

[177] Jingjing Ren, Martina Lindorfer, Daniel Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. "Bug Fixes, Improvements, ... and Privacy Leaks – A Longitudinal Study of PII Leaks Across Android App Versions". In: *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*. 2018.

[178] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. "ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic". In: *Proc. of the International Conference on Mobile Systems, Applications and Services (MobiSys)*. 2016.

[179] Martin Roesch et al. "Snort: Lightweight intrusion detection for networks." In: *Lisa*. Vol. 99. 1. 1999, pp. 229–238.

[180] Andrew Rosenberg and Julia Hirschberg. "V-Measure: A Conditional Entropy-Based External Cluster Evaluation Measure". In: *Proc. of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. 2007.

[181] Florian Roth and Thomas Patzke. *Sigma: Generic Signature Format for SIEM Systems*. 2017. URL: `https://github.com/SigmaHQ/sigma`.

[182] Delia Rusu, Blaž Fortuna, Dunja Mladenic, Marko Grobelnik, and Ruben Sipoš. "Document visualization based on semantic graphs". In: *2009 13th International Conference Information Visualisation*. IEEE. 2009, pp. 292–297.

[183] Brendan Saltaformaggio, Hongjun Choi, Kristen Johnson, Yonghwi Kwon, Qi Zhang, Xiangyu Zhang, Dongyan Xu, and John Qian. "Eavesdropping on Fine-Grained User Activities Within Smartphone Apps Over Encrypted Network Traffic". In: *Proc. of the USENIX Workshop on Offensive Technologies (WOOT)*. 2016.

[184] Kiavash Satvat, Rigel Gjomemo, and VN Venkatakrishnan. "EXTRACTOR: extracting attack behavior from threat reports". In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2021, pp. 598–615.

[185] Taneeya Satyapanich, Francis Ferraro, and Tim Finin. "CASIE: Extracting Cybersecurity Event Information from Text". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 05. 2020, pp. 8749–8757.

[186] Tara Seals. *Good Heavens! 10M Impacted in Pray.com Data Exposure*. Nov. 2020. URL: `https://threatpost.com/10m-impacted-pray-com-data-exposure/161459/`.

[187] Tara Seals. *Millions of Hotel Guests Worldwide Caught Up in Mass Data Leak*. 2020. URL: https://threatpost.com/millions-hotel-guests-worldwide-data-leak/161044/.

[188] Farsight Security. *Newly Observed Domains*. https://www.farsightsecurity.com. Jan. 2020.

[189] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "“Andromaly”: a behavioral malware detection framework for Android devices". In: *Journal of Intelligent Information Systems* (2012).

[190] Ravi Sharma. *Android Q will let you run multiple apps simultaneously with Multi Resume feature*. https://www.91mobiles.com/hub/android-q-multi-resume-feature-how-to-use-2-android-apps-at-same-time. Nov. 2018.

[191] Yun Shen, Enrico Mariconti, Pierre Antoine Vervier, and Gianluca Stringhini. "Tiresias: Predicting security events through deep learning". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018, pp. 592–605.

[192] Yun Shen and Gianluca Stringhini. "Attack2vec: Leveraging temporal word embeddings to understand the evolution of cyberattacks". In: *28th USENIX Security Symposium*. 2019, pp. 905–921.

[193] Xiaokui Shu, Frederico Araujo, Douglas L Schales, Marc Ph Stoecklin, Jiyong Jang, Heqing Huang, and Josyula R Rao. "Threat intelligence computing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018, pp. 1883–1898.

[194] Wee Meng Soon, Hwee Tou Ng, and Daniel Chung Yong Lim. "A machine learning approach to coreference resolution of noun phrases". In: *Computational linguistics* 27.4 (2001), pp. 521–544.

[195] Katy Stalcup. *AWS vs Azure vs Google Cloud Market Share 2020: What the Latest Data Shows*. Aug. 2020. URL: https://www.parkmycloud.com/blog/aws-vs-azure-vs-google-cloud-market-share/.

[196] Statcounter. *Mobile Browser Market Share Worldwide*. https://gs.statcounter.com/browser-market-share/mobile/worldwide. Accessed: February 2019.

[197] Statista. *Number of Apps Available in Leading App Stores as of 2nd Quarter 2019*. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. Aug. 2019.

[198] Tim Stöber, Mario Frank, Jens Schmitt, and Ivan Martinovic. “Who do you sync you are? Smartphone Fingerprinting via Application Behaviour”. In: *Proc. of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec).* 2013.

[199] Strom, Blake E. and Applebaum, Andy and Miller, Doug P. and Nickels, Kathryn C. and Pennington, Adam G. and Thomas, Cody B. “MITRE ATT&CK®: Design and Philosophy”. In: *MITRE.* Ed. by The MITRE Corporation. `https://attack.mitre.org/`, retrieved 2023-04-12. 2018.

[200] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. “Rethinking the inception architecture for computer vision”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR).* 2016, pp. 2818–2826.

[201] Duyu Tang, Furu Wei, Nan Yang, Ming Zhou, Ting Liu, and Bing Qin. “Learning sentiment-specific word embedding for twitter sentiment classification”. In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* 2014, pp. 1555–1565.

[202] Vincent F Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. “Robust Smartphone App Identification via Encrypted Network Traffic Analysis”. In: *IEEE Transactions on Information Forensics and Security* (2018).

[203] Vincent F. Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. “AppScanner: Automatic Fingerprinting of Smartphone Apps from Encrypted Network Traffic”. In: *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P).* 2016.

[204] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. “Feature-rich part-of-speech tagging with a cyclic dependency network”. In: *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics.* 2003, pp. 252–259.

[205] Kristina Toutanvoa and Christopher D Manning. “Enriching the knowledge sources used in a maximum entropy part-of-speech tagger”. In: *2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora.* 2000, pp. 63–70.

[206] Martin Ukrop, Lydia Kraus, Vashek Matyas, and Heider Ahmad Mutleq Wahsheh. “Will you trust this TLS certificate? perceptions of people working in IT”. In: *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC).* 2019, pp. 718–731.

[207] Minh Van Nguyen, Viet Dac Lai, Amir Pouran Ben Veyseh, and Thien Huu Nguyen. "Trankit: A light-weight transformer-based toolkit for multilingual natural language processing". In: *arXiv preprint arXiv:2101.03289* (2021).

[208] Eline Vanrykel, Gunes Acar, Michael Herrmann, and Claudia Diaz. "Leaky Birds: Exploiting Mobile Application Traffic for Surveillance". In: *Proc. of the International Conference on Financial Cryptography and Data Security (FC).* 2016.

[209] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. "FP-STALKER: Tracking Browser Fingerprint Evolutions". In: *Proc. of the IEEE Symposium on Security and Privacy (S&P).* 2018.

[210] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Advances in Neural Information Processing Systems (NIPS).* 2017, pp. 5998–6008.

[211] Nguyen Xuan Vinh, Julien Epps, and James Bailey. "Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance". In: *Journal of Machine Learning Research* (2010).

[212] Thomas Vissers, Jan Spooren, Pieter Agten, Dirk Jumpertz, Peter Janssen, Marc Van Wesemael, Frank Piessens, Wouter Joosen, and Lieven Desmet. "Exploring the ecosystem of malicious domain registrations in the .eu tld". In: *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID).* Springer. 2017, pp. 472–493.

[213] Sean Wallis. "Binomial confidence intervals and contingency tests: mathematical fundamentals and the evaluation of alternative methods". In: *Journal of Quantitative Linguistics* 20.3 (2013), pp. 178–208.

[214] Yong Wang, Jinpeng Wei, and Karthik Vangury. "Bring Your Own Device Security Issues and Challenges". In: *Proc. of the IEEE Consumer Communications and Networking Conference (CCNC).* 2014.

[215] Jonathan J Webster and Chunyu Kit. "Tokenization as the initial phase in NLP". In: *COLING 1992 volume 4: The 14th international conference on computational linguistics.* 1992.

[216] Lance Whitney. *2020 sees huge increase in records exposed in data breaches.* Jan. 2021. URL: `https://www.techrepublic.com/article/2020-sees-huge-increase-in-records-exposed-in-data-breaches/`.

[217] Chengcheng Xiang, Yudong Wu, Bingyu Shen, Mingyao Shen, Haochen Huang, Tianyin Xu, Yuanyuan Zhou, Cindy Moore, Xinxin Jin, and Tianwei Sheng. "Towards Continuous Access Control Validation and Forensics". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 113–129.

[218] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. "Detecting large-scale system problems by mining console logs". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*. 2009, pp. 117–132.

[219] Shuang Xun, Xiaoyong Li, and Yali Gao. "AITI: An automatic identification model of threat intelligence based on convolutional neural network". In: *Proceedings of the 2020 the 4th International Conference on Innovation in Artificial Intelligence*. 2020, pp. 20–24.

[220] Jia Yan, Ming Wan, Xiangkun Jia, Lingyun Ying, Purui Su, and Zhanyi Wang. "DitDetector: Bimodal Learning based on Deceptive Image and Text for Macro Malware Detection". In: *Proceedings of the 38th Annual Computer Security Applications Conference*. 2022, pp. 227–239.

[221] Hongyi Yao, Gyan Ranjan, Alok Tongaonkar, Yong Liao, and Zhuoqing Morley Mao. "SAMPLES: Self Adaptive Mining of Persistent LExical Snippets for Classifying Mobile Application Traffic". In: *Proc. of the ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*. 2015.

[222] YaraRules. *YaraRules Project*. 2015. URL: `https://yara-rules.github.io/blog/`.

[223] Ehtesham Zahoor, Zubaria Asma, and Olivier Perrin. "A formal approach for the verification of AWS IAM access control policies". In: *European Conference on Service-Oriented and Cloud Computing*. Springer. 2017, pp. 59–74.

[224] Ehtesham Zahoor, Asim Ikram, Sabina Akhtar, and Olivier Perrin. "Authorization policies specification and consistency management within multicloud environments". In: *Nordic Conference on Secure IT Systems*. Springer. 2018, pp. 272–288.

[225] Jun Zhao, Qiben Yan, Jianxin Li, Minglai Shao, Zuti He, and Bo Li. "TIMiner: Automatically extracting and analyzing categorized cyber threat intelligence from social data". In: *Computers & Security* 95 (2020), p. 101867.

[226] Ziyun Zhu and Tudor Dumitras. "Chainsmith: Automatically learning the semantics of malicious campaigns by mining threat intelligence reports". In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 458–472.

[227] Carson Zimmerman. *Ten strategies of a world-class cybersecurity operations center*. Tech. rep. Retrieved from `https://www.mitre.org/publications/all/ten-strategies-of-a-world-class-cybersecurity-operations-center` on 2020-10-26. 2014.

[228] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. "Why does your data leak? uncovering the data leakage in cloud from mobile apps". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1296–1310.