INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Implementation and evaluation of secure and scalable anomaly-based network intrusion detection

Philipp Mieden

Aufgabensteller:	Prof. Dr. Helmut Reiser
Betreuer:	Dipl.-Inform. Stefan Metzger Leibniz-Rechenzentrum München

Index terms— security, anomaly detection, intrusion detection systems

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den December 18, 2018

. . . .

(Unterschrift des Kandidaten)

Abstract

Corporate communication networks are frequently attacked with sophisticated and previously unseen malware or insider threats, which makes advanced defense mechanisms such as anomaly based intrusion detection systems necessary, to detect, alert and respond to security incidents. Both signature-based and anomaly detection strategies rely on features extracted from the network traffic, which requires secure and extensible collection strategies that make use of modern multi core architectures. Available solutions are written in low level system programming languages that require manual memory management, and suffer from frequent vulnerabilities that allow a remote attacker to disable or compromise the network monitor. Others have not been designed with the purpose of research in mind and lack in terms of flexibility and data availability. To tackle these problems and ease future experiments with anomaly based detection techniques, a research framework for collecting traffic features implemented in a memory-safe language will be presented. It provides access to network traffic as type-safe structured data, either for specific protocols or custom abstractions, by generating audit records in a platform neutral format. To reduce storage space, the output is compressed by default. The approach is entirely implemented in the Go programming language, has a concurrent design, is easily extensible and can be used for live capture from a network interface or with PCAP and PCAPNG dumpfiles. Furthermore the framework offers functionality for the creation of labeled datasets, targeting application in supervised machine learning. To demonstrate the developed tooling, a series of experiments is conducted, on classifying malicious behavior in the CIC-IDS-2017 dataset, using Tensorflow and a Deep Neural Network.

Contents

1 Introduction

Fueled by the invention of the TCP protocol (IP) in 1974 [CK74], modern life relies heavily on complex computer networks and applications. This ranges from everyday personal use of computers, to distributed business networks, medical and industrial applications and self driving cars. Due to the increasing amount of valuable information stored in such systems, they are becoming a more and more popular target for attackers. Successful attacks against computer systems often have a direct negative economic impact. Adversaries range from amateur enthusiasts, to organized criminals and state-sponsored organizations. [McN16] Due to the rising availability and sophistication of internet technology and the increased number of network attacks, network defense has become an important area of research. Network defense requires in-depth knowledge about all protocols flowing across the network. Capture software must therefore implement many complex parsers in order to interpret the collected data the right way. Keeping up with the increasing amount of data that has to be processed can be challenging, and requires effective implementations and algorithms. [Hos09] Real time data processing is necessary, to ensure a fast uncovering of an ongoing or past attack and to prevent or lighten economic damage. New vulnerabilities appear every day and are quickly exploited with so called zero-day attacks. Signature based detection strategies suffer from the inability to detect previously unknown threats. However, although having been a popular area of research over the last 20 years, machine learning techniques are rarely seen in commonly available tools. High amounts of false positives and the lack of proper training and evaluation data are well known problems of anomaly based intrusion detection. [Pat14] Traffic patterns change a lot - and so does the software stack inside network environments. To reflect this, modern datasets must be used for the evaluation of anomaly based intrusion detection strategies. Network intrusion detection greatly helps in identifying network breaches, tracing them back to the responsible parties and then taking action to isolate and retrieve any damage that occurred. The attack recognition and event monitoring capabilities of intrusion detection systems also have a deterrent effect on attackers, who face a greater risk of being discovered and prosecuted. [Bac00] The presence of an intrusion detection might convince an attacker to search for another target, that is easier to penetrate. Being blocked by the monitor or by an analyst due to an alert, creates unwanted attention for the intruder and slows down his operations. However, in order to benefit from intrusion detection, there is a need for a reliable and extensive data source in order to make accurate predictions.

1.1 Acknowledgements

Acknowledgements go to my supervisor Dipl.-Inform. Stefan Metzger and Prof. Dr. Helmut Reiser from the Leibniz Rechenzentrum Munich (LRZ), for offering me to write my thesis in cooperation with the institute, guiding me in my research, critical reflections of my ideas and many constructive discussions. To Vern Paxson and Aashish Sharma from the Lawrence Berkeley National Laboratory, for the interesting conversations at the Bro Workshop Europe 2018 in Karlsruhe, which inspired many of my design decisions for this research project. To Kaspersky Lab, for their annual student contests, which motivated me to conduct research on the topic of network intrusion detection, and publish my tooling to assist other researchers in the field. And finally, to my family, which thoroughly supported and encouraged me in my journey towards finishing my thesis, and contributed to my survival by frequently bringing food into my computer cave.

1.2 Outline Of The Thesis

This introduction explains the motivation for network security monitoring and anomaly based detection strategies, and outlines the impact of memory corruptions on the security of todays networking infrastructure. Common terminology will be explained, along with a problem definition and a task description to define the scope of this thesis. After a brief discussion of related work, requirements for network feature collection mechanisms are analyzed, followed by an evaluation of existing solutions. Afterwards, concept and implementation of the research project will be outlined in-depth, as well as ideas for future improvements and several use cases beyond network intrusion detection. In the final evaluation, several practical appliances of the developed tool are shown, which includes a series of experiments on classifying malicious behavior with a deep neural network in Tensorflow. After the conclusion, a look ahead on possible future developments is given.

1.3 Motivation

Preventative techniques like authentication and access control can fail, intrusion detection provides a second line of defense [BK14] and serves as the base for incident response after a system was compromised. [BCV01] After gaining a foothold on a network, time passes for the attacker to perform reconnaissance and identify assets of interest. Although the initial infection might not be detected, it is possible to prevent further damage when detecting the lateral movement of the attacker. Intrusion detection plays a critical role in uncovering such behavior and greatly aids in reconstructing it for further forensic investigation. Mandiant / FireEye published statistics from 2013, outlining an average time from an intrusion to incident response of more than 240 days. Only one third of organizations discovered the intrusion themselves. [Bej13] Their latest report from 2018 states a median detection time of 99 days in 2016 and 101 days in 2017. [mtr18] Even though the decreasing detection time is a positive development, it is still too high. 100 days are a lot of time for an attacker to identify assets of interest and take action. Besides that, even the most sophisticated and secure systems are still vulnerable to insiders, who misuse their privileges. [Den87]

When inspecting vulnerability visualizations, an alarming trend becomes visible: the overall number of reported vulnerabilities each year is growing and the vast amount of them is scored as medium or high severity. Figure 1.1 shows a distribution of vulnerabilities and their different severity rankings (low, medium and high) over the last years. The rankings are based upon the CVSS V2 Base score, more details and visualizations can be found on the NVD CVSS page. [nvd18]

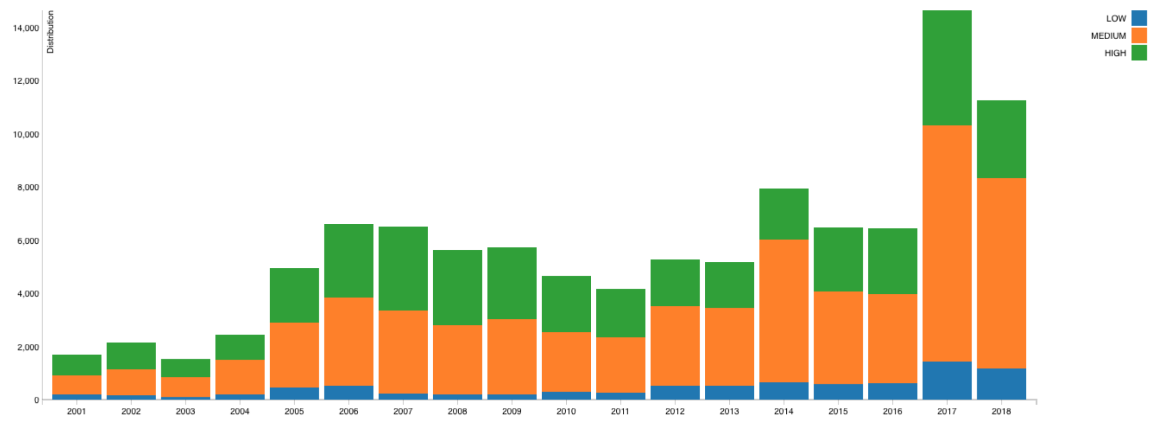


Figure 1.1: CVSS Severity Distribution Over Time

Intrusion detection is a key component for network defense and information security, it helps identifying attempts to compromise information systems and protect their availability, integrity, and confidentiality. [MSA05] Security information and event management (SIEM) systems often struggle with organizing the huge amount of data and provide interoperability between various data formats and input sources. Data fed into a SIEM can be divided into 4 categories: network traffic, host data from the monitored endpoints, logs generated by various systems and threat intelligence feeds.

Network data is the most valuable of these informations, since every attack has to go through the network and thus leaves traces, even if an attacker tries to hide or obfuscate his presence. On the contrary for example, host data is not always reliable, because a compromised machine can be manipulated by an attacker. A recent study from the American Consumer Institute Center for Citizen Research has shown that 83% of routers contain vulnerable code. Infrequent updates cause the devices to be vulnerable for a long period of time, even after the vulnerabilities have been published. [aci18] Although recent claims about hardware attacks by implants in the supply chain of the server manufacturer supermicro have not been proven at the time of this writing, they describe a possible scenario that should be considered when securing a network environment. Network security monitoring can detect malicious attempts to contact the outside world, from both compromised software or hardware components, and allows to search the gathered data for indicators of compromise afterwards, to determine if there were other attempts in the past. In order to exfiltrate assets or information, the data has to be transferred to a server that is controlled by the intruder. This makes network data a very powerful piece of evidence. A recent study on the history of Remote Access Tools (RATs) [rat18] shows that the development of trojans has not stopped, instead it seems to be increasing. The study analyzed the 300 most important RATs of the last 20 years. It shows the importance of behavior based detection strategies, since signatures can only identify known malicious programs, and new malware variants appear frequently.

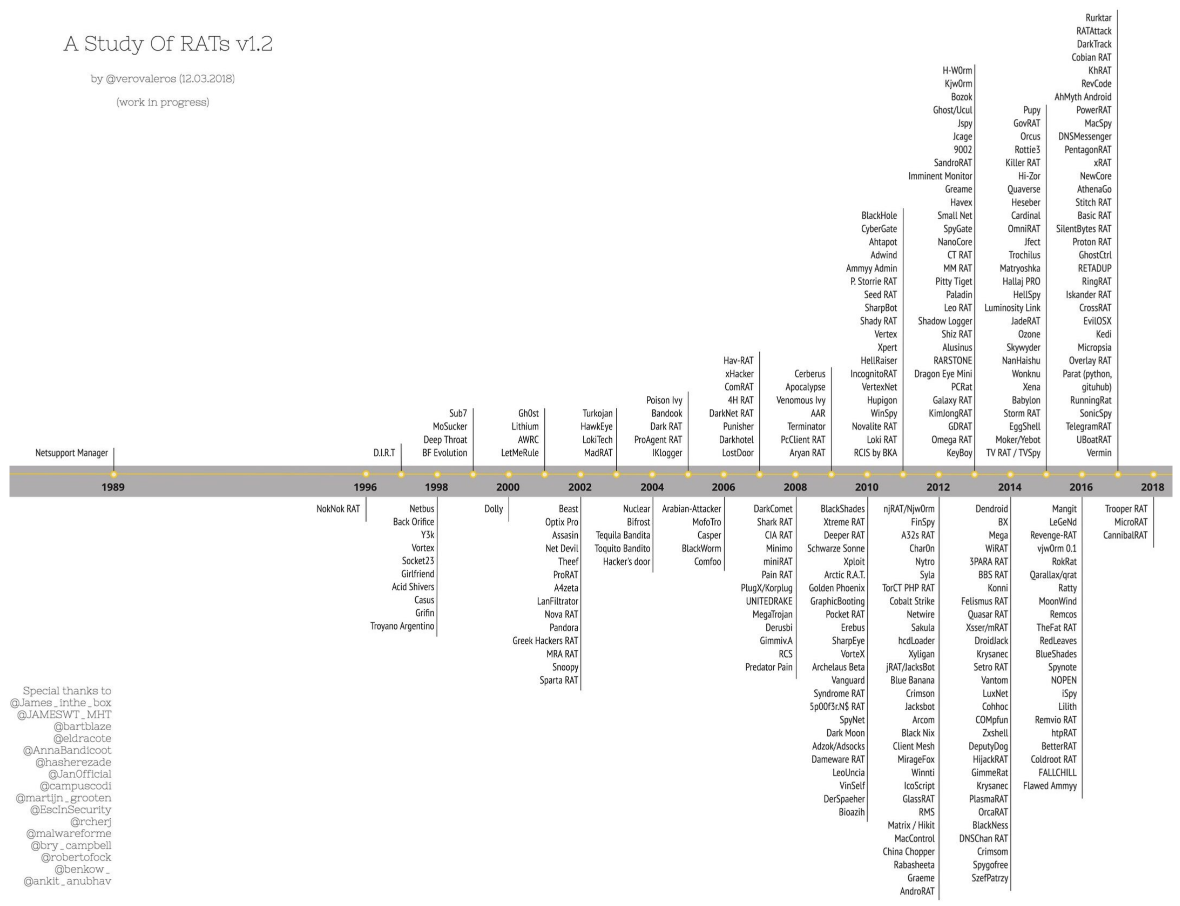


Figure 1.2: RATs Study v1.2

Many software systems are exploitable, due to logic errors or memory corruption vulnerabilities. An example are buffer overflows, in which a section of memory is being overfilled with data and the following section overwritten, which can lead to a crash or even remote code execution. Another common example are use after free vulnerabilities, which refer to the process of accessing memory after it has been freed. This can also cause a program to crash or result in the execution of arbitrary code. The C programming language and its derivatives without automated memory management, suffer from these kinds of vulnerabilities, due to human errors in bounds checking or even because of compiler optimizations that eliminate bound checks. [cap17]. However, also languages that provide such automated memory management are not prune to attacks. For example, the Java virtual machine [jav11] and object serialization [jav17] have been exploited in the past. Web browsers are a well known target for attacks, as they implement a huge protocol stack and have many potentially vulnerable components, such as plugins and extensions. A commonly exploited component of web browsers is Adobe Flash, a programming language to create animated web contents.

Even big companies that invest large amounts of money in a security team and offer bug bounties for submitted vulnerabilities, fail to protect themselves or their users from attacks on a regular basis. In the recent facebook incident, attackers exploited a technical vulnerability to steal access tokens that would allow them to log into about 50 million people's accounts. [fb218] Apple recently fixed a vulnerability in the ICMP packet error handling code of their XNU kernel, which resulted in a heap overflow and could potentially be exploited for remote code execution (CVE-2018-4407). The vulnerability exists in such a fundamental part of the operating systems networking code that anti-virus software is unable to detect exploitation attempts. [apl18] Also, even companies run by engineers aware of computer security implications, such as the italian spyware manufacturer HackingTeam, have been compromised with a zero-day exploit against an embedded device in their network in the past. [hac18] A new zero day (CVE-2018-15454) in Cisco's Adaptive Security Appliance and Firepower Threat Defense Software has recently been discovered, due to an error in handling Session Initiation Protocol (SIP) packets. The vulnerability allows an attacker to remotely reload or crash the affected systems. [cis18] The attack surface is further increased by the tools used for forensic investigation. Protocol dissectors have a huge attack surface, due to the amount of supported protocols and implemented code that has be to maintained. Wireshark for example, has 1400 authors, supports over 2400 protocols and is made of a total of at least 2.5 million lines of code. [wir18] Due to its complexity and dynamic nature, current software stacks are potentially vulnerable to exploitation and compromise. Monitoring them is therefore necessary to ensure rapid uncovering of breaches and to increase the attackers efforts and costs. It is important to keep in mind that also the network monitor might be the target of an attack.

For the following tables the MITRE CVE database has been queried, it holds a total of 108021 entries at the time of this writing. The results listed here should be taken with a grain of salt: not all issues are severe or remotely exploitable, and some queries deliver results that did not affect the search term itself but a software product running on it (especially Operating System queries). Also some queries deliver inaccurate results, due to keywords present in descriptions, i.e. the 'Golang' search returns four results, but one result is a vulnerability in the Joomla framework which is written in PHP. In this case it is listed because the proof of concept exploit is written in Golang and this is mentioned in the description. Language keyword searches contain results of vulnerabilities in products that are written in the language, but the vulnerability arised from an error in the product and not in the used language. However, this provides a rough estimation of the problem dimension and the attack surface. All search queries have been submitted at the 10th of october 2018. An interesting observation is that 5143 CVE entries match the 'memory corruption' keywords, which means that almost 5% of the collected vulnerabilities are related to memory corruption. The search term "overflow" delivers 11417 results, which makes almost 11% of all tracked vulnerabilities.

OS	CVEs
Windows	6317
Linux	4905
macOS	2010
iOS	2325
Android	4788

Table 1.1: CVE results for commonly used operating systems

Browser	CVEs
Firefox	2260
Chrome	1796
Opera	357
Internet Explorer	2051
Safari	1029

Table 1.2: CVE results for common browsers

Format	CVEs
JSON	213
BSON	7
MessagePack	0
YAML	48
ProtoBuf	1
SOAP	193
XML	1227

Table 1.3: CVE results for common serialization formats

Vendor	CVEs
Netgear	78
Asus	68
LinkSys	84
Juniper	259
HPE	312
D-Link	226
TP-Link	104
ZyXEL	68
Cisco	3869

Table 1.4: CVE results for common router manufacturers

Language	CVEs
Java	1828
Javascript	2060
Adobe Flash	1363
PHP	6003
.NET	1298
C++	65
Golang	3
Haskell	0
Rust	3

Table 1.5: CVE results for commonly used programming languages

Bugs or vulnerabilities exist across the whole technology stack that is being used in co-operate or personal environments. The risk that attackers will find a way in is high, and depends on the amount of time and money adversaries are willing to spend. Vulnerabilities can potentially be abused to disable, disrupt or circumvent the monitoring system. The network monitor might even serve as an entry point to the network it was supposed to protect, and thus weakens the overall system security, instead of increasing it. To shed some light on the situation regarding the most popular open source projects, various searches have been conducted and will be summarized below. Four databases for vulnerabilities have been queried using their search function, for the keywords *snort*, *suricata* and *bro*, namely the MITRE CVE, NIST NVD, vulDB and exploitDB. If configurable in advanced search, the exact match option has been selected to filter the results. Without this, a search query for *bro* would also deliver results for entries containing the keyword *browser*. For vulDB the product name field was used to improve search accuracy. The estimated number of unrecorded cases is probably much higher, since many bugs get discovered and fixed by the authors without being assigned a CVE identification. Additionally, software libraries that are commonly used are at risk of being exploited and increase the attack surface, for example *libpcap* or *bpf*; they will be included in the search as well. Many of these vulnerabilities occur due to memory corruptions, such as buffer under- or overflows and heap corruptions. With proper auditing and unit testing, logic errors can be detected early in the development cycle.

To estimate the amount of unreported incidents, a search in git log history of the projects version control system (git) was conducted, the following command was used to filter the log entries and display the result:

```
git log --grep="memory-leak" \
        --grep="memleak" --grep="memory leak" \
        --grep="memory corruption" --grep="memory-corruption" \
        --grep="CVE" \
        --grep="denial-of-service" --grep="denial of service" \
        --grep="overflow" \
        --grep="out of bounds" --grep="out-of-bounds" \
        --pretty=oneline | wc -l
```

For searching the linux kernel source for netfilter and bpf git history, the last line needs to be changed to filter for paths that contain the respective names, example for bpf:

```
...
--pretty=oneline *bpf* | wc -l
```

For *snort*, the github repository of *snort3* was queried. VulDB displays 50 entries at a maximum for unpaid users, *libpcap* and *netfilter* delivered more results than that, and thus were marked with 50+.

Finally a google search was conducted that used the following search pattern. Enclosing the search keywords with double quotations requires an exact match on these phrases, multiple statements have been combined with a logical OR, newlines have been added for readability. The ␣ character represents a whitespace, framework is a placeholder for the search term.

```
"<framework> security vulnerability" OR
"<framework> buffer overflow" OR
"<framework> memory corruption" OR
"<framework> memory leak" OR
"<framework> denial of service"
```

The results are displayed in table 1.6 Because the amount of possible vulnerabilities should be seen in relation to the age of the framework, the year of creation was added as well.

	snort	suricata	bro	libpcap	bpf	netfilter
created	1998	2009	1994	1999	1992	1998
nvd	42	13	0	3	31	59
cve	38	12	4	3	24	60
vulDB	25	13	3	50+	28	50+
exploitDB	27	1	0	0	11	5
repository	44	162	165	24	126	168
google	407	10	8	5	9	232

Table 1.6: Potential security vulnerabilities in IDS and their components

This search was performed on the 31th of August 2018. The *bro* repository was queried for the git log history at last commit 33a8e7, *suricata* last commit was 8c7aee, *linux* kernel source last commit was 420f51, *libpcap* last commit was 96571e, *snort3* last commit was f3268ef4f.

It should be noted that the authors of those frameworks are highly experienced software engineers, who have a strong sense for security and their code is audited by several similar experienced developers. Nevertheless, mistakes happen and can lead to compromise or denial-of-service attacks if not discovered and patched in time. Many of the vulnerabilities could have been avoided by choosing a memory safe language instead of using C for better performance.

Running the git log query for potential vulnerabilities on the latest linux kernel source yields 12297 results, while a CVE search for linux returns only 4817 results. This confirms the assumption that the dark digit of potential vulnerabilities is much higher than the amount being tracked in vulnerability databases.

The report "State of Software Security 2018" from the security company veracode [Ver18], represents the results of analyzing over 2 trillion lines of code across 700,000 scans, conducted over a one year period between April 1, 2017 and March 31, 2018. This gives an impression regarding the dimension of the attack surface in modern software environments:

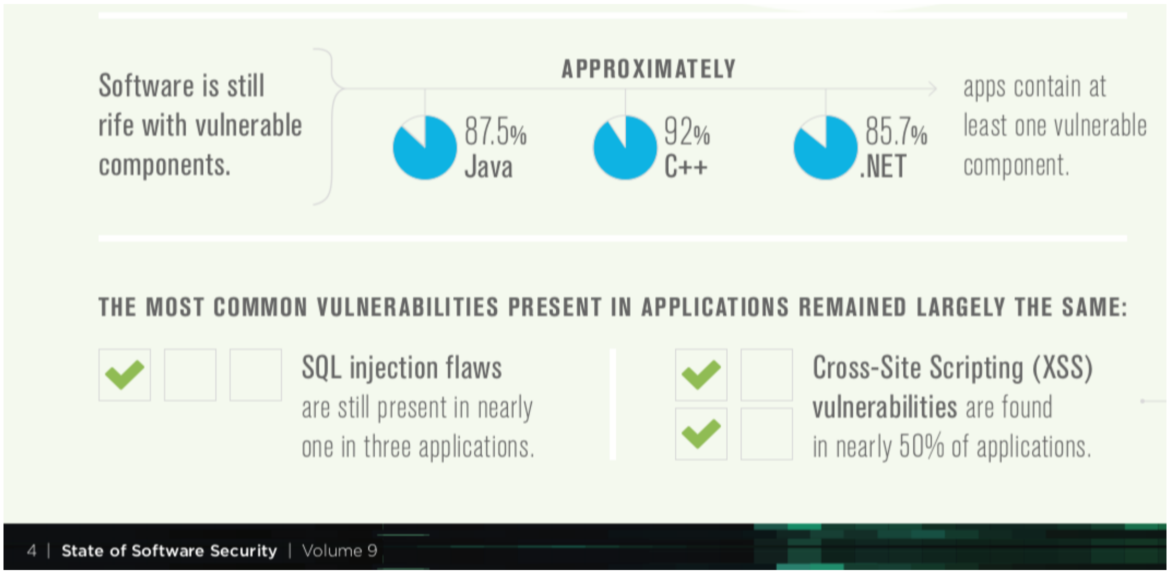


Figure 1.3: State of Software Security 2018 excerpt

Code is often written in a hurry, as developers need to get things done on time. This introduces programming mistakes and bad design decisions, which can lead to exploitable vulnerabilities. Source code audits are expensive and time intensive and therefore often omitted. Even more problematic is the sanitation of a production infrastructure. Changes can lead to new bugs or malfunctions that negatively impact the systems availability. Penetration testing on the live environment is often not possible, which increases costs because a testing environment has to be created for this purpose. The risk that an intruder, who invests sufficient amounts of time and money, will always be able to successfully penetrate an IT infrastructure, has to be taken into account. Therefore secure and efficient strategies for detecting system compromise are necessary to protect todays network environments.

1.4 Terminology

When extracting artifacts from network traffic several different terminologies are commonly used: collection, selection, extraction and generation. Because these terms are often confused or mixed up, the following section outlines their differences and characteristics.

1.4.1 Data Collection

Data collection describes the task of building up a dataset from an input data source. In the collection stage as much information as possible should be yielded, to provide a rich data set for the selection or extraction phases. A simple example for a collected feature would be an IP address from an IP packet.

1.4.2 Feature Extraction

Also referred to as feature generation, this describes techniques for obtaining new features from existing ones. Because this requires further computations and therefore consumes a lot of system resources, extracted features must be chosen carefully and their contribution to the detection task must be evaluated. Example for extracted features include: Inter packet arrival times, average flow size, payload entropy etc. Feature extraction methods are transformative, the data is projected into a new feature space with lower dimension. Examples include Principal Component Analysis (PCA) and Singular Value Decomposition (SVD). [BK14]

1.4.3 Feature Selection

Feature selection refers to the task of choosing features from a dataset. This usually happens after gathering all data and aims to reduce the amount of data that has to be processed by the machine learning algorithm. Data that does not sufficiently contribute to the detection task is removed, as well as highly redundant or correlated data. The dimensionality of the input feature set is reduced by selecting a subset of the original features. [BK14] The answer to the question which features are the most relevant depends on various factors, such as network size and architecture, used machine learning algorithms and the goal of the network monitoring operation. For example, the purpose of the network monitor could be software debugging, security operations or network performance analysis.

1.5 Problem Definition

Feature collection for the purpose of network intrusion detection faces several problems. First of all, the increasing volume of data that has to be processed: high resolution video streaming, large file transfers and video conferences are common scenarios in corporate environments, and generate huge amounts of network packets that have to be processed. Another problem are attacks against the monitoring system, which can result in crashes and therefore loss of audit data, or in the worst case even to remote compromise of the network monitor. [Pax98] Current solutions for collecting features from traffic are written in low level system programming languages. This increases the attack surface tremendously, since these languages do not provide automated memory management. This can lead to memory corruption vulnerabilities that enable denial of service or remote code execution attacks. Even if the state of an implementation is considered audited and secure, future updates to protocol specifications or new protocols will require continuous modifications, which can possibly (re-) introduce exploitable security vulnerabilities. Furthermore, the collection sensor should not be dependent on a specific architecture and be functional across all major platforms, in order to qualify for a widespread area of applications. The output data format should be consumable by as many systems as possible, to enable easy and quick integration into an existing network monitoring stack. The most commonly used data format for big data analysis and machine learning are comma separated values (CSV), which do not preserve data types or provide structure to the data. Deployment and configuration of existing solutions is complex and time intensive for large environments. A growing share of internet traffic is encrypted, denying access to content and requiring analysis strategies that do not rely on cleartext packet payloads for classification. The current state of tooling is insufficient to meet requirements for convenient and effective experiments, due to the fact that most of those tools were not designed specifically with a research task in mind. Core questions of anomaly based intrusion detection haven't been solved completely, after more than two decades of research: Which are the most powerful features, which do require extraction and which don't? Which feature combination delivers the best results across all available datasets or algorithms? Which features are delivering the best results for a specific algorithm? What is the most efficient summary structure for the intrusion detection task? What delivers better results, observations for each protocol or summary structures such as flows and connections? Which technique is the most efficient in terms of computing resources? Solving these questions entirely requires efficient tooling for feature collection, selection and extraction. Unfortunately, many researchers don't publish their tools, which makes it hard to reproduce and validate their results, and increases time needed for future research on the topic. Furthermore, relying on crafted datasets for verification of network security monitoring, does not reflect the actual situation inside the protected network environment. There might be fundamental differences in architecture or in client behavior. Collecting network traffic during a penetration test inside the network and using this as a dataset to verify the deployed countermeasures work as expected, would create a much more realistic scenario and therefore more meaningful results. Currently, no publicly available tooling exists to accomplish this kind of independent verification in an effective and reproducible manner.

Modern:

- Programming lanuage with safe guards
- Ouput format that preserves type safety and data structure
- Utilize multiple processor cores with scaling ability
- Good documentation
- Anomaly based classification strategy
- Provide all data prior to generating abstractions

1.6 Task Description

Key context ->

Goal of this thesis is the implementation of a modern, systematic approach to access network protocol specific information, for the purpose of research on anomaly based intrusion detection strategies. In this context, the term modern refers to the following attributes of the implemented framework: A suitable programming language must be chosen for the implementation, which supports all major platforms, provides memory safety and has a concurrent programming model. A suitable output format must be selected that preserves type safety and structure of the data, while being platform neutral. To make use of modern computer architecture with multiple processor cores and allow scaling for large workloads, the implementation must have a concurrent design. Also it must be well documented to support other researchers working with it in the future. The implemented approach should be suitable for usage with dump files and live traffic from a network interface and provide a command-line and library interface. Furthermore, unlike many solutions that try to achieve maximum data reduction, this approach shall focus on making the data more accessible and provide maximum flexibility, in order to improve the workflow and efficiency for experiments. This follows the philosophy, that a network feature collection system should provide data at the lowest level of its internals, prior to generating any abstractions on top of it. The implemented approach shall be evaluated with an up-to-date dataset and an anomaly based classification strategy.

1.7 Related Work

The following sections briefly describes related work, ordered by its year of publication.

1.7.1 Literature

Michael W. Lucas [Luc10] presents several network traffic flow capture formats, tools and techniques to visualize, filter and analyze network flow data.

Richard Bejtlich [Bej13] covers the general practice of network security monitoring and incident response, and provides lots of background knowledge on network topologies and sensor placement. Common tools are explained and their usage is demonstrated.

Bhattacharyya et al [BK14] present several feature selection algorithms among classification techniques to identify network anomalies and vulnerabilities at various layers. They also cover the assessment of network anomaly detection systems, present different tools and discuss research challenges.

Michael Collins [Col17] presents techniques for experimental data analysis, traffic behavior analysis, data collection using sensors and network mapping using python. He also deals with application identification and provides various ways to visualize network data.

1.7.2 Articles

Vern Paxson [Pax98] describes requirements for a network intrusion detection system and explained the structure and functionality of the bro network monitor. Although this paper is more than 20 years old at the time of this writing, it contains fundamental aspects of network security monitoring, and inspired this research project with its philosophy of separating mechanism from policy.

Felix Iglesias et al. [IZ14] reduce the 41 features widely adopted from the KDD dataset, to only 16 features, while still achieving similar detection results as with the full feature set. They used several filter algorithms including Weight by Maximum Relevance (WMR), Stepwise Regression and Stability Selection. For validation the Bayes Classifier was used, besides Support Vector Machines (SVM). This research paper served as a motivation for questioning the need for overly complex extracted features, and served as a hint to conduct experiments with a more basic feature set.

Kim et al [KKD$^+$15] presented their packet extraction tool for large volume network traces and compared the performance to existing solutions. Unfortunately, while their implementation could have been of great interest to the research community, they do not seem to have released it. The fact that their results could therefore not be evaluated, and that future research will need to continue with inefficient tooling, greatly encouraged me to publish my whole tool chain, in order to assist future researchers and allow independent verification of my results.

Chakchai So-In [Si16] analyzed the state of the art for traffic flow analysis, and provided an overview and comparison of common tools and capture formats. It served as a useful introduction to flow based collection methodology.

2 Requirement Analysis

In the following, software requirements for feature collection from network traffic are defined and explained. Weighting the requirements has been intentionally omitted. The more of these characteristics are offered by a particular implementation, the better it is suited for network monitoring. An ideal solution should fulfill all requirements. Any missing requirement will cause trouble at one point in the development or usage cycle.

2.1 Functional Requirements

2.1.1 Protocol Support Coverage

The feature collector must be able to parse as many protocols as possible, in order to get the most detailed view on activity on the network. Additionally the system must report on traffic that it does not understand.

2.1.2 Data Availability

For research purposes, the network monitor should make the collected data available at the lowest possible level, prior to generating summary structures. That means, protocol specific information should be offered for every protocol that can be interpreted by the monitor.

2.1.3 Abstraction Capabilities

Using the information that has been gathered in the collection stage, abstractions, summary structures, aggregate values and statistics can be generated from the collected data. A popular example for summary structures are unidirectional network flows and bidirectional connections.

2.1.4 Concurrent Design

With the evolution of multi-core processors and their decreasing cost, more and more systems are equipped with this technology. When designing a monitoring system, availability of several processing cores should be taken into account from the beginning. The term concurrency refers to the process of splitting a big problem into many small sub-problems, which unveils opportunities to execute independent operations in parallel. Concurrency describes programming as the composition of independently executing processes. Parallelism refers to programming as the simultaneous execution of, possibly related, computations. While concurrency could also be expressed as dealing with lots of things at once, parallelism can be described as doing lots of things at once. While concurrency refers to structure, parallelism refers to execution. Although not identical, both are related. In order to enable concurrency, independent executions must be coordinated through communication. [rob18] Since receiving network packets always happens sequentially, no matter if reading them from a dump file or live from the network card, parallelizing their decoding process is an option to utilize the power of multi-core systems, and speed up the overall processing. Many systems haven't been designed with concurrent processing in mind from the ground up.

2.1.5 File Extraction Capabilities

Various network protocols allow the transfer of files or have been specifically created for this purpose. Examples include HTTP, HTTPS, FTP, SFTP and BitTorrent. Other protocols can be abused to carry file payloads as well. A prominent example for covert data exfiltration is the DNS protocol. Encrypted communication such as HTTPS also makes inspection impossible, without access to the used certificate or cryptographic key. From a security monitoring perspective, files sent across a network need to be extracted to perform further analysis, like matching executables against anti-virus (AV) databases. Breaking encryption for this purpose should not be considered, because this weakens the overall system security, which is not a desirable effect.

2.1.6 Supported Input Formats

Suitable formats for offline network data input need to be supported, for this purpose the industry standards PCAP and the improved version PCAPNG are the best candidates. PCAP preserves every single packet and the full payloads, and thus is a suitable input format for data collection. However, this comes at the cost of increased size of the dump files, which is especially problematic when storing traffic over long periods of time. For this reason, other formats such as Cisco's NetFlow have been developed, that summarize traffic on a per flow basis without payload data, which greatly reduces size.

2.1.7 Suitable Output Formats

The output format of the collected data should not only be consumable by many different systems, libraries and frameworks, but also displayable in a human readable format. Comma separated values (CSV) is a commonly chosen output format for this, it is not limited to a particular character set, and work just as well with Unicode as with ASCII. However CSV do not preserve the character set currently in use, this must be communicated separately. Although human readable in its original form, CSV is not a space efficient mechanism of storing or exchanging information. Additionally, CSV cannot represent hierarchical or object-oriented structured data, without separating sub structures from its parent context.

2.1.8 Real-Time Operation

Live capture from a network interface should be supported, to allow operation and monitoring in real time. Since dumping the collected data to disk is not always an option due to disk space constraints, e.g. on embedded devices, capture to disk should be optional and an alternate data collection mechanism should be provided. Uncovering successful breaches in real time and raising alerts, puts pressure on attackers to move quicker in a compromised network. This increases the possibility for mistakes and leaves less time for cleaning up traces, which is beneficial for damage reduction, incident response and digital forensics. A system that is too slow might generate alerts with a high delay, thus giving attackers more time to accomplish their goals. Identifying attempted breaches in real time gives network security operators valuable information about targeted systems and used techniques, and helps to put counter measures into place.

2.2 Non-Functional Requirements

2.2.1 Memory Safety

When parsing untrusted input, it must be guaranteed that there are no exploitable vulnerabilities in the parser code. Since modern network communication is based on a huge stack of protocols, many different and potentially very complex parsers have to be implemented and maintained, which tremendously increases the chances of programming errors in the corresponding parser code. Code audits are often omitted due to high cost, or only performed on a fraction of the available source code. Additionally, many vendors violate the RFC specifications for protocols, which requires parsers that are fault tolerant and have been tested with fuzzing extensively. Choosing a language with a memory safe garbage collected runtime for the design of a feature collection system, allows to focus on creating robust parsing logic rather than secure memory handling.

2.2.2 Open Source Codebase

The source code should be accessible and well documented, to enable other researchers to audit or extend the code. Although this makes the code also available to potential attackers, the security model of the network monitor must not rely on secret code.

2.2.3 Scalability

The system should be scalable to meet requirements for large scale distributed deployments, such as receiving input data from many sensors across the network. This should happen in an efficient and secure way, that means data should be compressed when being transferred, and the network monitor must encrypt its communication to prevent eavesdropping.

2.2.4 Performance

The number packets of that has to be captured and processed, is rising continuously. The network monitor must be efficient enough to keep up with all the data, without loosing anything. Although an alert does not automatically prevent an intrusion, it is a first step towards detection, and increases the risk for an attacker to be discovered. An overloaded system that starts to drop packets, can lead to missing the discovery of an ongoing attack. Performance depends heavily on the programming language used for the implementation. As mentioned before, low level system programming languages do not provide memory safety. Because of this, the usage higher level languages such as rust, haskell or Golang for the design of security critical infrastructure, should be considered in the future. This introduces a performance penalty, however this is a trade-off that comes with great benefits for the overall application security. A recently published paper implemented a POSIX compliant kernel in Golang, and measured a performance penalty ranging from 5-15% in comparison to the C implementation. [CKM18] Todays networks transport lots of huge traffic streams, for example video streaming such as netflix, torrent downloads and more. Because these flows cause a huge performance degradation for the monitoring system, a common practice is to detect those so called 'elephant flows' and discard them at the earliest stage possible. [SSK15] However, this imposes the security risk of missing relevant information, an attacker could use the knowledge of certain traffic types being disarded to hide in a big data stream. Since audio and video players and their codecs are a common source for security vulnerabilities, discarding these flows may lead to missing an actual intrusion, due to an exploited vulnerability in the media player. Searching the MITRE CVE database for the VLC media player for example returns 92 results. An optimal implementation should not discard any data to meet performance goals.

2.2.5 Configurable Design

The monitor application needs to be configurable, to allow performance tweaking and adaption to special requirements, such as for example privacy regulations. Automated capture and analysis of network traffic can result in severe privacy violations, and may violate local, state and national laws. Therefore it is necessary to obtain the required permissions for the target network as well as qualified legal advice, prior to installing any monitoring software. How to protect the users privacy during a network security monitoring operation has become an important question for many network administrators. [Bej13] Since the 28th of May 2018, the General Data Protection Regulation (GDPR) further increases protection of personal data from citizens of the European Union, by drastically increasing the fines for violations as well as the scope of protected information.

2.2.6 Extensibility

Computer networks are highly dynamic environments and thus require the network monitor to be easily extensible, to integrate new protocols as fast as possible and keep up with the extremely fast development of technology. For this reason, the monitoring system should allow to add parsers for new protocols in a fast and secure manner, and should provide an interface to further extend functionality.

2.2.7 Reliability

The monitor must always be available and deliver meaningful results, even under attack or in high load scenarios. Also, critical errors must not lead to a shutdown of the monitoring system, but should notify the administrator and rollback the system to working state. Recovering from system failure, no matter if due to logic errors or due to a fault produced by an attacker, it is important to ensure continuous monitoring. Reporting and recovering on parsing errors or undecodable traffic is mandatory, because the monitor will be subject to attacks.

2.2.8 Usability

The user interface of the network monitor should be powerful enough to accomplish complex tasks, but also be simple enough to be efficiently used by a human, without confusion or waste of time. Since network data is very abstract for humans, the software should provide proper visualization options, to assist humans in understanding, correlating and analyzing huge amounts of high dimensional data in a short period of time.

2.2.9 Storage Efficiency

Storage is limited and the continuously increasing amount of data that modern networks are transporting increases the amount of collected audit data. Therefore, the generated output should be as small as possible, while preserving all information relevant for the detection task. Additionally it should be configurable what data will be collected. In some scenarios there might be data that is not relevant for the detection task or a concrete monitoring system. If this is the case, no resources should be wasted on collecting or storing irrelevant data.

2.3 Summary

Feature collection systems for intrusion detection should run continually without or with minimal human supervision, and should be reliable enough to run in the background of the observed system and impose minimal overhead. The system must be fault tolerant and able to recover from errors automatically, and should preserve its own integrity in order to detect attacks against the monitor itself. Also important are aspects like reconfigurability and extensibility, since dynamic environments such as computer networks often require adaptations. Deployment should be easy and fast and the system must be able to observe deviations from normal behavior. [Pat13] Additionally, they must provide an efficient user interface and provide visualization to assist human analysts. The named requirements for the underlying hardware also play a vital role in ensuring the monitoring systems integrity and performance. For future reference, all software requirements for modern network intrusion detection systems have been listed in table 2.2

Requirement	Short Description
Protocol Support Coverage	Range of supported protocols
Data Availability	Access to data on the lowest level
Abstraction Capabilities	Generation of abstractions such as flows
Concurrent Design	Efficient use of multi-core architectures
File Extraction	Extraction of files from network traffic
Suitable Output Format	Widely supported and efficient output format(s)
Real Time Operation	Live acquisition of traffic from a network interface
Supported Input Formats	Support for industry standard input formats
Memory Safety	Secure memory management
Open Source Codebase	Publicly available source code
Scalability	Operation in large scale, possibly distributed networks
Performance	Efficient processing for high volume traffic
Configurable Design	Configuration and adaption options for special requirements
Extensibility	Ease of extension for new protocols
Reliability	Reliable implementation
Usability	Ease of use for the analyst / researcher
Storage Efficiency	Low storage overhead

Table 2.2: Summary of feature collection requirements

3 State Of The Art

This section describes the most popular formats and tools that can be used to gather information about network traffic. They are listed in no particular order. Snort (written in C++) and suricata (written in C) do not seem to provide an interface to export collected information from the network traffic. Therefore, they are not suited for the purpose of gathering features for anomaly detection. All lines of code statistics have been generated using the cloc tool (*github.com/AlDanial/cloc*) in version 1.80.

3.1 Flow Formats

In general, a flow record is defined as a group of packets with a number of common properties, from a certain time interval. [Pat14] A series of packets that share the same IP protocol, source and destination IP addresses and ports, is called a five-tuple IP flow. Flow records have a small footprint and do not include the data exchanged between two systems, but still deliver valuable insights into a network. Even for large scale networks, flow records over several years only result in several hundred gigabytes of allocated disk space. A TCP connection creates two flows: one from client to server and one from server to client. [Luc10] Flow-based IDS (FIDS) cannot detect attacks related to packet payload, for example the ping of death attack does not create a change in flow frequency and has a low traffic volume. The attack makes use of large ICMP packets that violate the specification of ICMP (maximum size: 56 bytes), but not the specification of the IP protocol, which allows a packet size of up to 65535 bytes. If not handled properly by the receiving application, this can lead to a buffer overflow when reassembling the fragmented packet. Flows are instead suited to detect attacks such as denial-of-service (DoS) attacks, scans, worms and botnet communication. [Pat14]

3.1.1 NetFlow

NetFlow was developed by Cicso and served as a convenient way of monitoring network communication and data reduction. Routers that support NetFlow can export data via UDP or SCTP to NetFlow collectors. Records contain timestamps for flow start and end, number of bytes and number of packets observed, source and destination addresses, type of service, TCP flags and more. [Si16]

3.1.2 sFlow

An industry standard technology for monitoring high speed switched networks, developed as a competitor to NetFlow. It provides insight into the utilization of networks, enables performance optimization, allows for usage billing and counter measures against security threats. When embedded within switches or routers, application level traffic flows can continuously be monitored at wire speed on multiple interfaces simultaneously. [sFl03]

3.1.3 IPFIX

Originally developed by the IETF, the Internet Protocol Flow Information Export (IPFIX) is an open standard for exporting flow information from routers or other probes. Based on Cisco NetFlow Version 9, the standard defines how IP flow information has to be formatted and transferred from an exporter to a collector, the protocols requirements were outlined in the original RFC 3917. It is a push protocol, each probe will periodically transfer IPFIX messages to configured receivers. [ipf18]

3.2 Packet Level Formats

3.2.1 PCAP

The PCAP format saves a complete dump of the network packet data to disk for offline analysis. Each dump file has a PCAP file header, with some meta information about the capture, followed by the captured packets. Each network packet consists of a packet header with the timestamp and packet length, followed by the complete packet data. Because PCAP includes the packets full payloads, it requires lots of disk space. It is the output format of the wireshark protocol dissector tool. [DH12] For anomaly based intrusion detection PCAP in its original form is an unsuitable format, as it does not provide the data in a structured form and requires manual parsing and processing to extract relevant information.

3.2.2 PCAP-NG

The PCAP Next Generation Dump File Format (or PCAPNG for short) is an attempt to overcome the limitations of the currently widely used (but limited) PCAP format. The PCAPNG file format specification is still work in progress. PCAPNG supports captures from multiple interfaces in one file, improved timestamp resolutions, embedding of comments and additional meta data in the capture file, and was designed with extensibility in mind. [pca18]

3.3 Data Collection Tools

3.3.1 Argus

Argus is a system for network audit record generation and utilization, which focuses on monitoring of large scale networks. It generates enriched network flow data and operates on dump files or on the wire, and was designed to support network operations, performance monitoring and security management. It can be used to monitor a network in real-time or check captured data against indicators of compromise. The *Argus* sensor processes packets and generates very detailed network flow reports, focusing on efficient storage, processing and inspection of large amounts of network traffic. Many metrics, such as connectivity, availability, reachability, loss, jitter and retransmission, are provided for all observed network flows. Attributes from packet contents, including mac addresses, tunnel identifiers, protocol ids and options are preserved as well. *Argus* is an open source project written in the C programming language, and currently supports macOS, Linux, Windows (under Cygwin), FreeBSD, OpenBSD, NetBSD, Solaris, AIX, HP-UX, VxWorks, IRIX and OpenWrt. It has been embedded in network adapters and ported to various hardware accelerated platforms. As of version 3.0.8.2, the argus server consists of 28610 lines of C source and 10141 lines of

C headers, while the argus client of the same version has 111401 lines of C source code and 19526 lines of C headers.

3.3.2 nProbe

Developed by nTop, *nProbe* is a NetFlow v5 and v9 and IPFIX probe and collector that supports IPv4 and IPv6. Commonly used in commercial environments, it can be used to gather and export flows information generated by devices on the network. It can serve as a replacement for embedded, low-speed, NetFlow probes. Analysis of networks with a traffic volume of multiple gigabytes can be monitored at full speed, with only a moderate packet loss under heavy loads. Collected flows can be sent to a collector, such as the open source ntopng or a commercial solution. *nProbe* is written in C and its source code is not publicly available. However, ntop offers *nProbe* free of charge to universities.

3.3.3 Bro / Zeek

Bro is an open source traffic monitoring system with intrusion detection capabilities, that passively monitors traffic and generates events, which can then be dynamically handled in the turing complete bro scripting language. It aims at decoupling mechanism from policy, that means it simply describes the traffic it sees, leaving all assumptions and interpretation to be defined by the analyst depending on the current scenario. *Bro* is implemented in C++ and was originally created by Vern Paxson in 1994 as a research project for the Berkeley National Lab. The name 'Bro' is an orweillian reference to the fact that network security monitoring goes hand in hand with severe privacy violations. Events generated by Bro are policy neutral and require being handled by a script to take action, like alerting the administrator by mail or executing a system command. *Bro* logs information about events and flow information, SSL handshakes, SSH handshakes and protocol specific data. It provides analyzers for the most common application layer protocols such as: HTTP, FTP, SMTP and DNS. [Pax98] *Bro* supports all major platforms but was not designed with concurrency in mind. It has been extended to work in a cluster, to tackle the need for large scale packet analysis. Parsers are not handwritten anymore, the bro authors developed a parser generation framework (binPAC), to ease extension and maintenance of protocol parsers and reduce attack surface. However, also with binPAC several memory related bugs are documented in the security changelog of the bro website [bro18b]. The authors are currently working on a new version of the parser generator, named SPICY [SAH16]. SPICY uses HILTI, an abstract execution environment for concurrent, stateful network traffic analysis [SVDCP14]. After being announced in 2014, the project website still states that both HILTI and SPICY are currently in prototype state, and not production ready yet. As of commit 3f206cb8a, *Bro* consists of 81288 lines of C++ source code and 19402 lines of C++ header definitions. Of all evaluated solutions, *Bro* is by far the most advanced and should be considered when monitoring large scale networks. The project was recently renamed to *Zeek*.

3.3.4 CICFlowMeter

CICFlowMeter (previously known as ISCXFlowMeter) is a research project from the Canadian Institute for Cybersecurity. It serves as a data generator for anomaly detection and has been used in various cybersecurity datsets from the CIC. These include the IPS/IDS dataset (CICIDS2017), Android Adware-General Malware dataset (CICAAGM2017) and Android

Malware dataset (CICAndMal2017). The application is written in Java and generates rich bidirectional flow information, with more than 80 features, such as packet size statistics for both directions and total appearances of packets with certain flags set. As of commit 1d4e34e, the *CICFlowMeter* project contains 5083 lines of Java code and 1875 lines of C headers.

3.3.5 ipsumdump

Ipsumdump uses *libpcap*, or packet sockets on Linux, to read IP packets from the network, or from a PCAP dump file, and writes an ASCII summary of the packet data to the standard output. Added comments on the first couple of lines describe the contents of the generated summary. It uses the click modular router, a C++ engine to process network packets from the same author. *Ipsumdump* is written in C, as of commit 1903f1c, the project contains 39626 lines of C++ source and 20631 lines of C headers.

3.3.6 tshark

Tshark is the command-line interface to the popular wireshark packet dissector and provides a huge set of options and configurations. It behaves similar to the tcpdump tool, but provides wireshark's filtering capabilities and access to all of wiresharks dissectors. *Tshark* is is written in C, as of commit 1f9414a, it consists of 2885 lines of C code. The last commit on the *tshark.c* source file, 1f9414a on Sep 4 2013, fixes a memory leak.

3.4 Requirement Evaluation

In the following, previously discussed requirements will be briefly evaluated for each of the presented current solutions.

Argus

Argus delivers data in its own summary structure flow format and currently lists 65 available fields in its wiki. For parallel processing worker threads are used, file extraction is not supported. Data can be exported as CSV, the system can be deployed for real-time monitoring. PCAP and PCAPNG are supported as input formats. The *Argus* source code for both client and collector is publicly available and licensed under the GNU General Public License version 3. [Bej13] *Argus* can be scaled to meet the requirements for distributed networks. Usability is improved by offering several command-line tools for working with the generated data. The tool can be configured with various command-line flags and a configuration file. *Argus* client tools support reading compressed argus data, however it seems that argus does not compress the output by default when generating it.

nProbe

nProbe provides IPFIX or NetFlow information. It is written in C++ and does not seem to offer file extraction. Real time operation is supported and data can be exported as CSV. PCAP and PCAPNG are supported input formats, *nProbe* can be scaled to meet the requirements for distributed networks, configuration is possible with various command-line flags and a configuration file. Output written to disk does not seem to be compressed by default, data in transit however is being compressed.

Bro / Zeek

Zeek does provide extensive data for each protocol, and a connection log that serves as a summary of network activity. It does not have a concurrent design, but features a parallel approach: processing can be achieved by running several instances in a cluster and using their broker for communication. It meets the requirements for monitoring large scale networks. *Zeek* does provide partial memory safety, by using a parser generator, instead of hand written parsing code. However, memory corruptions can still occur in the remaining parts of the application and also the parser generation had security relevant bugs in the past. Also noteworthy is the fact that the parsers currently do not seem to be tested with fuzzing on a regular basis, a ticket in the bro issue tracker from 2011 brings up the topic but was closed unresolved after a while. [bro18a] It does not provide a native way to convert its output into CSV, but a python script (github.com/red8383light/BRO2CSV) exists that can take care of the conversion. *Zeek* can extract transferred files from various protocols and generates a separate file log. Real time operation is supported as well as PCAP and PCAPNG as input formats. The *Zeek* Source code is publicly available. Usability is good, the authors put a lot of effort in making the data accessible, for example by using the bro-cut tool to select fields of interest. *Zeek* can be configured with various command-line flags and a configuration file. Output logs consist of lines with tab separated values and output is written to disk without compression.

CICFlowMeter

CICFlowMeter provides only bidirectional flow information, no access for individual protocols. The tool uses several worker threads to process incoming packets in parallel. It can export data as CSV and operate live on a network interface. *CICFlowMeter* does not provide file extraction capabilities, PCAP and PCAPNG are supported input formats, it does not seem to support monitoring distributed networks. Configuration with command-line flags is possible. The tool is written in Java and should in theory support all major platforms. However, it depends on a PCAP processing library (jnetpcap) that is not officially supported on macOS. Jnetpcap seems to be a Java wrapper for *libpcap*, the official web presence of the project (jnetpcap.com) was not reachable at the time of this writing. Java is known for numerous severe security problems in serialization and its vulnerable virtual machine, as outlined in the introduction and in the CVE comparison for different programming languages in table 1.5 and figure 1.3. The *CICFlowMeter* source code was briefly examined and findings will be discussed in the following. Numerous comments in the source files *PacketReader.java* and *TrafficFlowWorker.java* mention an unfixed *BufferUnderflowException* while decoding a header, that should have been solved by making deep copies of the packets, but apparently wasn't. Besides these comments, the source code is almost completely undocumented. A glimpse at the corresponding github repository reveals problems when parsing large PCAP files: finished flows are continuously collected in memory until the complete file was processed. Currently tracked flows is also continuously growing because flows are not correctly timed out. A pull request that fixes this was added on May 16, 2018. No reaction to this from the authors at the time of this writing.

Some users are complaining that the UDP Flows are incomplete. The source code is publicly available on github and as of commit 1d4e34e, it contains only 368 lines with comments of a total 4519 of lines of Java source code in *src/main/java/cic*. A majority of these comments is commented-out code instead of descriptive comments.

ipsumdump

Ipsumdump can read compressed PCAP files, and does not offer file extraction. Data can be exported as CSV, and packets can be read live from the network interface. The source code is publicly available, PCAP and PCAPNG are supported as input formats, there is no functionality for monitoring distributed networks. Accessing protocol specific information is possible with various command-line flags.

tshark

Tshark allows to access individual protocol fields, flow and stream information can also be accessed. Files can be extracted by supplying a command-line flag, and data can be exported as CSV. Printing specific fields for the desired protocols is possible, but tedious. Real time operation is supported, PCAP and PCAPNG are supported as input formats. The *Tshark* source code is publicly available as a single file in the *wireshark* repository. It is not suited for continuously monitoring large scale distributed networks. Usability is not optimal, there is a huge amount of options and commands for extracting values of interest can grow many lines long.

3.5 Summary

A majority of the existing solutions was not specifically designed for research on anomaly based intrusion detection techniques. The only exception from that, *CICFlowMeter*, fails to meet requirements regarding protocol access and platform support. While many of the available tools offer parts of the desired functionalities, none provides all, or offers them in a convenient way for research on anomaly based detection techniques. To deliver data in a way that provides value to the researcher, many additional pre- and post-processing steps have to be performed. None of the existing solutions is written in a memory safe language. A majority does not make the gathered data available for each protocol. No tooling exists to create labeled datasets. Current tooling is either overly complex and therefore time consuming to extend and debug, or limited in functionality and poorly documented. By not providing access to each protocol seen on the wire, many solutions obscure details, that could have been of interest to the researcher. Overall the situation of current tooling for experiments on anomaly based intrusion detection, is very unsatisfying and needs improvement.

The following table summarizes the requirement evaluation.

Requirement	Argus	nProbe	Zeek	ipsumdump	tshark	CICFlowMeter
Protocol Support Coverage	y	y	y	y	y	y
Data Availability	p	n	y	p	y	n
Abstraction Capabilities	y	y	y	y	y	y
Concurrent Design	y	y	n	n	n	y
File Extraction	n	n	y	n	y	n
Suitable Output Format	p	p	y	p	p	p
Real Time Operation	y	y	y	y	y	y
Supported Input Formats	y	y	y	y	y	y
Memory Safety	n	n	p	n	n	p
Open Source Codebase	y	n	y	y	y	y
Scalability	y	y	y	n	n	n
Performance	y	y	y	y	y	x
Configurable Design	y	y	y	y	y	p
Extensibility	x	x	y	y	p	p
Reliability	x	x	y	x	x	x
Usability	y	x	y	p	p	x
Storage Efficiency	p	n	n	n	n	n

Table 3.2: State of the art requirements evaluation

```
Legend:
y = yes, requirement fulfilled.
n = no, requirement not fulfilled.
p = requirement only partially fulfilled.
x = could not be evaluated
```

With respect to the requirement of memory safety, the only candidate for a possible extension, would have been *CICFlowMeter*. However, the before mentioned substantial security problems of the Java programming language, lack of documentation for the tool and the incompatibility of the PCAP processing library with the development system (macOS), led to the decision of creating a new implementation.

4 Concept

NETCAP

Traffic Analysis Framework

The *Netcap* (NETwork CAPture) framework efficiently converts a stream of network packets into highly accessible type-safe structured data that represent specific protocols or custom abstractions. These audit records can be stored on disk or exchanged over the network, and are well suited as a data source for machine learning algorithms. Since parsing of untrusted input can be dangerous and network data is potentially malicious, implementation was performed in a programming language that provides a garbage collected memory safe runtime. *Netcap* uses Google's *Protocol Buffers* to encode its output, which allows accessing it across a wide range of programming languages [pro18]. Alternatively, output can be emitted as comma separated values (CSV), which is a common input format for data analysis tools and systems. The developed tooling is extensible and provides multiple ways of adding support for new protocols, while implementing the parsing logic in a memory safe way. It provides high dimensional data about observed traffic and allows the researcher to focus on experimenting with novel approaches for detecting malicious behavior in network environments, instead of fiddling with data collection mechanisms and post processing steps. It has a concurrent design that makes use of multi-core architectures. The name *Netcap* was chosen to be simple and descriptive. The command-line tool was designed with usability and readability in mind, and displays progress when processing packets. In the following, *Netcap*'s design goals and high level concepts will be presented, along with a specification.

4.1 Design Goals

Core design goals of *Netcap* include:

- memory safety when parsing untrusted input
- ease of extension
- output format interoperable with many different programming languages
- concurrent design
- output with small storage footprint on disk
- maximum data availability
- allow implementation of custom abstractions
- rich platform and architecture support

The following graphic shows a high level architecture overview:

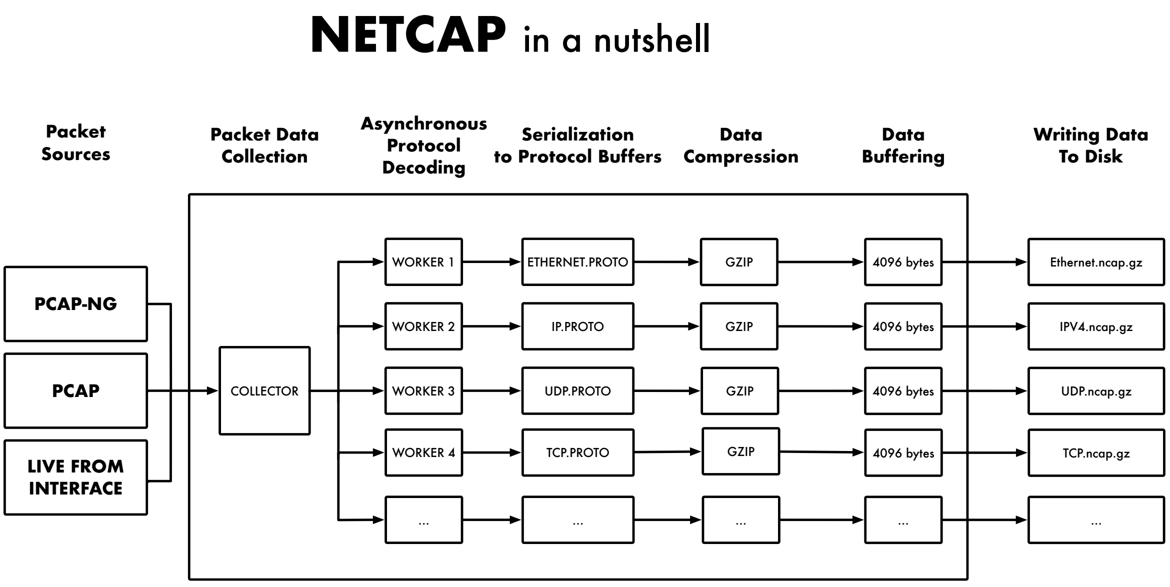


Figure 4.1: Netcap high level design overview

4.2 Netcap Specification

Netcap files have the file extension *.ncap* or *.ncap.gz* if compressed with gzip and contain serialized protocol buffers of one type. Naming of each file happens according to the naming in the *gopacket* library: a short uppercase letter representation for common protocols, and a camel case version full word version for less common protocols. Audit records are modeled as protocol buffers. Each file contains a header that specifies which type of audit records is inside the file, what version of *Netcap* was used to generate it, what input source was used and what time it was created. Each audit record should be tagged with the timestamp the packet was seen, in the format *seconds.microseconds*. Output is written to a file that represents each data structure from the protocol buffers definition, i.e. *TCP.ncap*, *UDP.ncap*. For this purpose, the audit records are written as length delimited records into the file.

4.3 Protocol Buffers

Google's *Protocol Buffers* are a platform neutral mechanism for serializing structured data. Originally developed for remote procedure calls (RPC), it provides ease of use and good performance, compared to other binary serialization formats. The *Protocol Buffer* compiler converts the *.proto* definitions into code for the target language. Officially supported are: *C++, Java, Python, Objective-C, C#, JavaScript, Ruby, Go, PHP, Dart.* However, the *protoc* compiler has a plugin architecture and many third-party plugins exist for other languages, including: *Erlang, D, R, Haskell, C, .NET, Perl, OCaml, Rust, Scala, Swift, Vala, Lua, Matlab, Prolog, Julia, Elm, Elixir, Delphi, Closure* and more. *Netcap* uses the protocol buffers v3 specification.

4.4 Delimited Protocol Buffer Records

The data format on disk consists of gzipped length-delimited byte records. Each delimited *Protocol Buffer* record is preceded by a variable-length encoded integer (*varint*) that specifies the length of the serialized protocol buffer record in bytes. A stream consists of a sequence of such records packed consecutively without additional padding. There are no checksums or compression involved in this processing step.

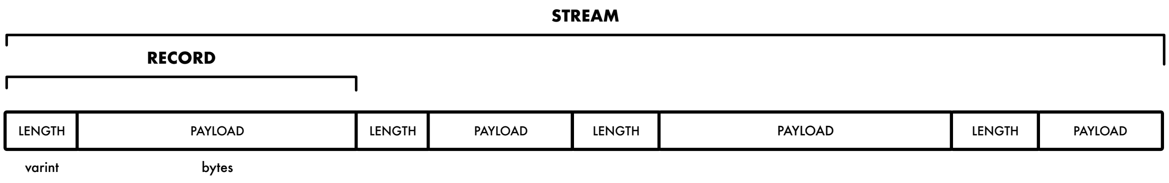


Figure 4.2: Netcap delimited data

4.5 Data Pipe

The *Netcap* data pipe describes the way from a network packet that has been processed in a worker routine, to a serialized, delimited and compressed record into a file on disk.

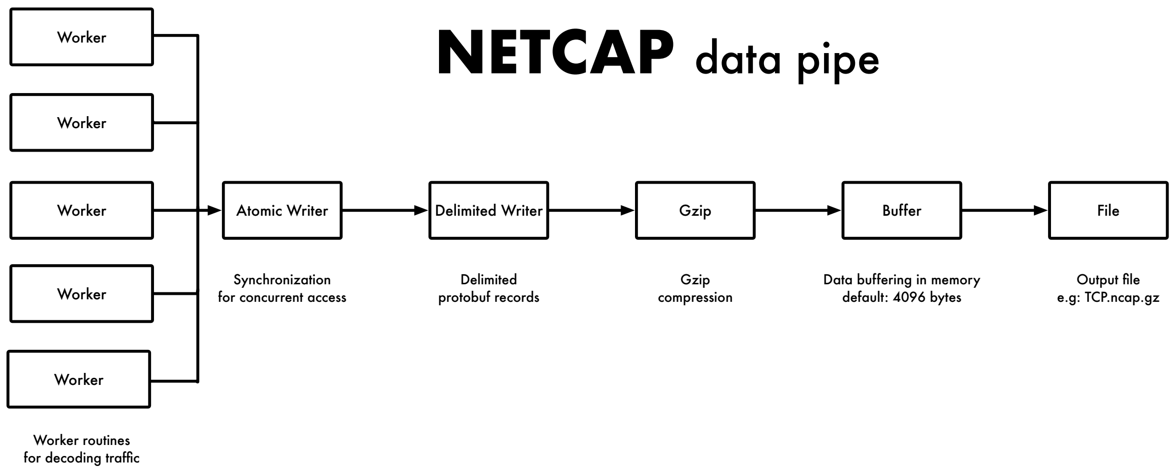


Figure 4.3: Netcap data pipe

4.6 Parallel Processing

To make use of multi-core processors, processing of packets should happen in an asynchronous way. Since *Netcap* should be usable on a stream of packets, fetching of packets has to happen sequentially, but decoding them can be parallelized. The packets read from the input data source (PCAP file or network interface) are assigned to a configurable number of workers routines via round-robin. Each of those worker routines operates independently, and has all selected encoders loaded. It decodes all desired layers of the packet, and writes the encoded data into a buffer that will be flushed to disk after reaching its capacity.

4.7 Data Compression

Encoding the output as protocol buffers does not help much with reducing the size, compared to the CSV format. To further reduce the disk size required for storage, the data is gzipped prior to writing it into the file. This makes the resulting files around 70% smaller. Gzip is a common and well supported format, support for decoding it exists in almost every programming language. If this is not desired for e.g. direct access to the stored data, this can be toggled with the *-comp* command-line flag.

4.8 Writing Data To Disk

Writing data to disk can happen asynchronously for each audit record type, since the data is written into separate files. To reduce the overhead imposed on the system by frequent syscalls for writing data to disk, *Netcap* determines the block size of the filesystem on startup (*encoder/layerEncoder.go*), and uses this value as a size for its internal buffers. The block size represents the maximum number of bytes that can be written to the filesystem with a single syscall. Using this value as a size for the buffers drastically speeds up writing data to disk. Because each audit record type is written to a separate file, reading them back for analysis purposes can be implemented in parallel, in order to speed up processing time.

4.9 Audit Records

A piece of information produced by *Netcap* is called an audit record. Audit records are type safe structured data, encoded as protocol buffers. An audit record can describe a specific protocol, or other abstractions built on top of observations from the analyzed traffic. *Netcap* does currently not enforce the presence of any special fields for each audit record, however by convention each audit record should have a timestamp with microsecond precision. A record file contains a header followed by a list of length-delimited serialized audit records. Naming of the audit record file happens according to the encoder name and should signal whether the file contents are compressed by adding the **.gz** extension.

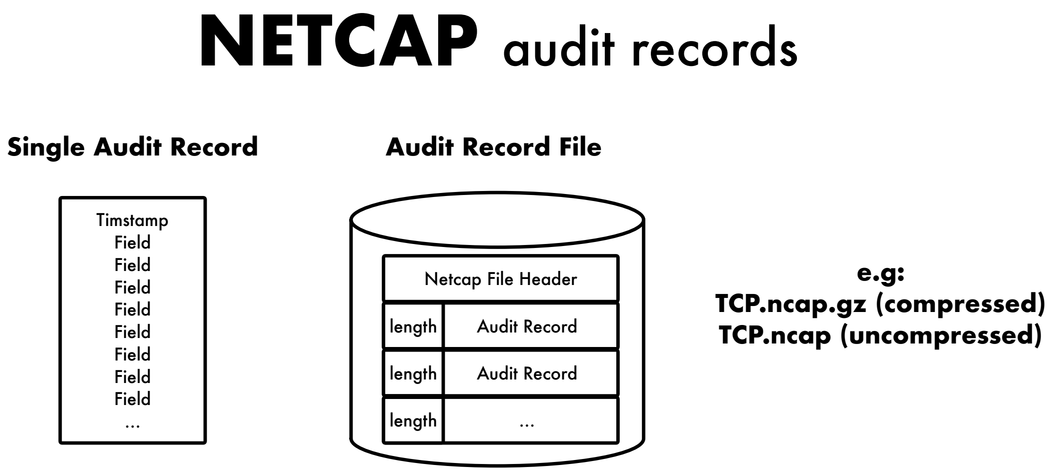


Figure 4.4: Netcap audit records

4.10 Netcap File Header

Each *Netcap* protocol buffer dump file, has a *Header* (type definition in *netcap.proto*) as its first element. The header contains information about the creation date, *Netcap* version, input source and the data type of the audit records.

```
type Header struct {
        Created                 string
        InputSource             string
        Type                    Type
        Version                 string
}
```

Type enumerations are maintained in the *netcap.proto* definitions. Due to the *C++* scoping implemented by the *Protocol Buffer* compiler, enumeration names cannot be the same as the corresponding message type. To solve this, a *NC_* prefix is prepended to each entry (NC stands for NetCap). Constants will follow the naming scheme *Type_NC_RecordName* in the generated code, for example the TCP constant is named: *Type_NC_TCP*.

```
enum Type {
        NC_Header                       = 0;
        NC_Batch                        = 1;
        NC_Flow                         = 2;
        NC_Connection                   = 3;
        NC_LinkFlow                     = 4;
        NC_NetworkFlow                  = 5;
        NC_TransportFlow                = 6;
        NC_Ethernet                     = 7;
        NC_ARP                          = 8;
        NC_Dot1Q                        = 9;
        NC_Dot11                        = 10;
        ...
}
```

4.11 Packet Decoding

For decoding each layer of the traffic, the *gopacket* library is used, which provides packet processing capabilities for Go. It is owned by Google and maintained by a community of developers. [gop18] The *gopacket* library is well documented and provides a mature API and abstractions for creating or decoding network packets. It offers support for reading and writing PCAPs and fetching packets live from an interface. The *gopacket* library offers an easy interface to exchange encoders for implemented protocols, or add support for new ones and is frequently updated by its developers. Using it for *Netcap* brings immediate support for the most common protocols and makes it easy to implement new ones, or extract additional data from implemented protocols. Choosing *gopacket* as a primary solution for decoding network data will lead to a growing list of supported protocols in the future. For the experiments and development of *Netcap*, version v1.1.15 of *gopacket* was used, from the master branch of the github repository, latest commit ec90f6c.

4.12 Workers

Workers are a core concept of *Netcap*, as they handle the actual task of decoding each packet. *Netcap* can be configured to run with the desired amount of workers, the default is 1000, since this configuration has shown the best results on the development machine. Increasing the number of workers also increases the number of runtime operations for goroutine scheduling, thus performance might decrease with a huge amount of workers. It is recommended to experiment with different configurations on the target system, and choose the one that performs best. Packet data fetched from the input source is distributed to a worker pool for decoding in round robin style. Each worker decodes all layers of a packet and calls all available custom encoders. After decoding of each layer, the generated protocol buffer instance is written into the *Netcap* data pipe. Packets that produced an error in the decoding phase or carry an unknown protocol are being written in the corresponding logs and dumpfiles.

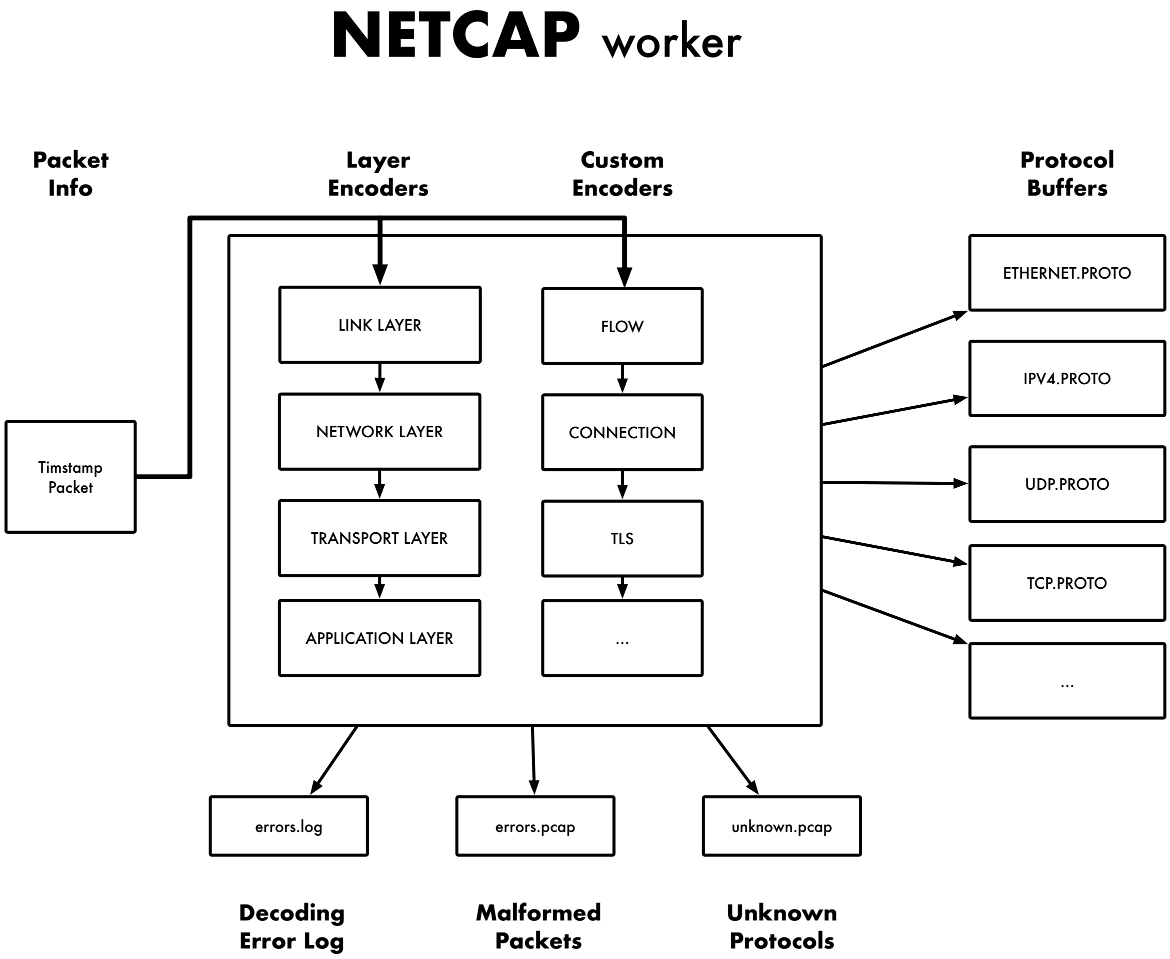


Figure 4.5: Netcap worker

Each worker receives its data from an input channel. This channel can be buffered, by default the buffer size is 100, also because this configuration has shown the best results on the development machine. When the buffer size is set to zero, the operation of writing a packet into the channel blocks, until the goroutine behind it is ready for consumption. That means, the goroutine must finish the currently processed packet, until a new packet can be accepted. By configuring the buffer size for all routines to a specific number of packets, distributing packets among workers can continue even if a worker is not finished yet when new data arrives. New packets will be queued in the channel buffer, and writing in the channels will only block if the buffer is full.

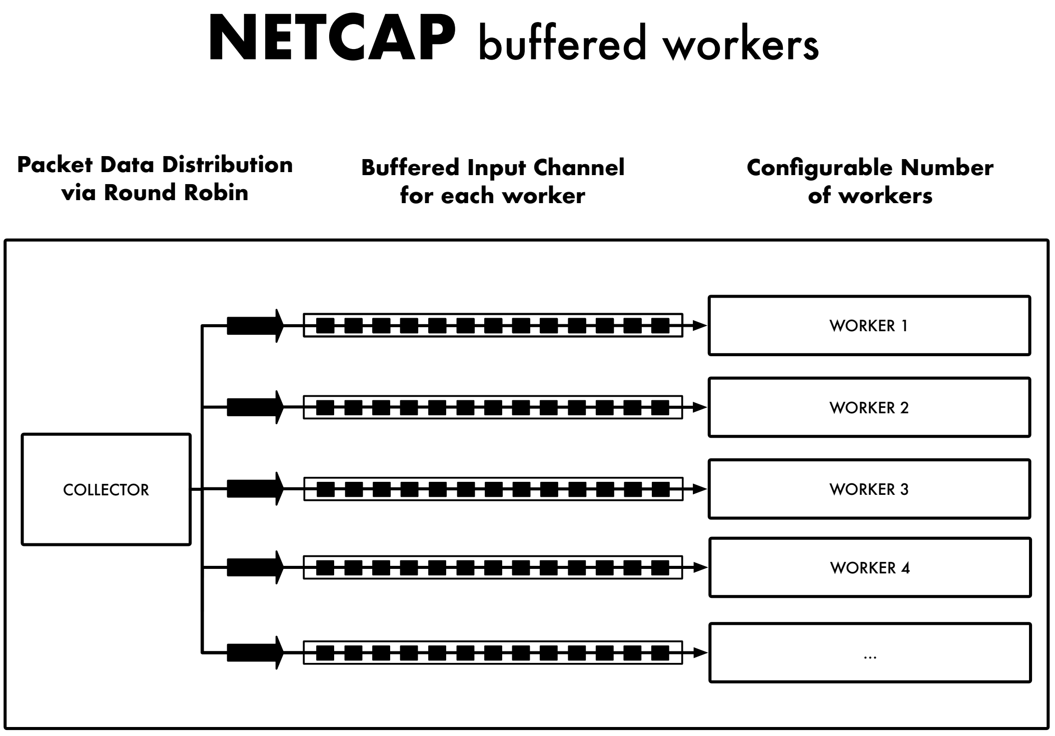

NETCAP buffered workers
Packet Data Distribution via Round Robin
Buffered Input Channel for each worker
Configurable Number of workers
COLLECTOR
WORKER 1
WORKER 2
WORKER 3
WORKER 4
...

Figure 4.6: Netcap buffered workers

4.13 Encoders

Encoders take care of converting decoded packet data into protocol buffers for the audit records. Two types of encoders exist: the *Layer Encoder*, which operates on *gopacket* layer types, and the *Custom Encoder*, for which any desired logic can be implemented, including decoding application layer protocols that are not yet supported by gopacket or protocols that require stream reassembly.

Layer encoder example:

The ethernet encoder takes the decoded packet, uses the available routing information and calculates the Shannon Entropy for the payload data.

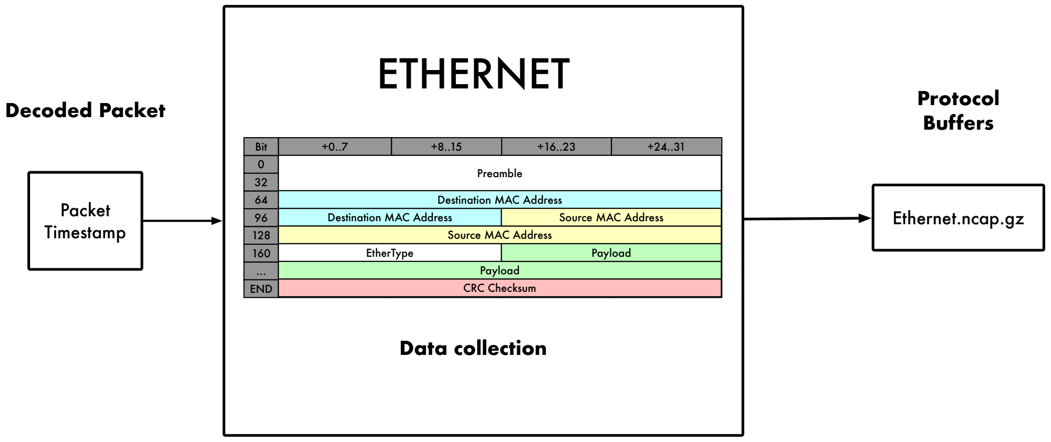


Figure 4.7: Netcap ethernet layer encoder

Custom encoder examples:

The HTTP decoder implements IPv4 stream reassembly, in order to extract HTTP requests and responses from both reassembled streams. Optionally, files from HTTP responses can be extracted and written to disk.

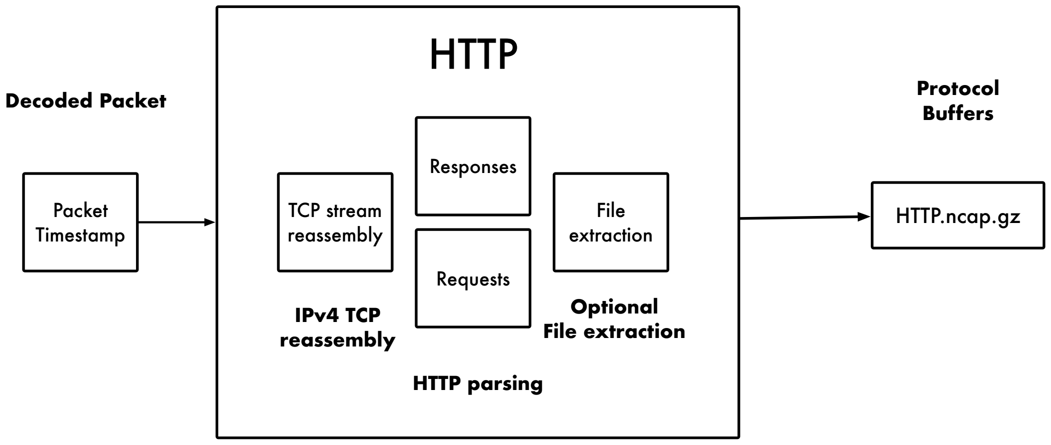


Figure 4.8: Netcap HTTP layer encoder

The *Connection* encoder holds a thread-safe connection store, to keep track of all seen connections and their properties. It is continuously flushed in configurable interval, in order to reduce memory usage.

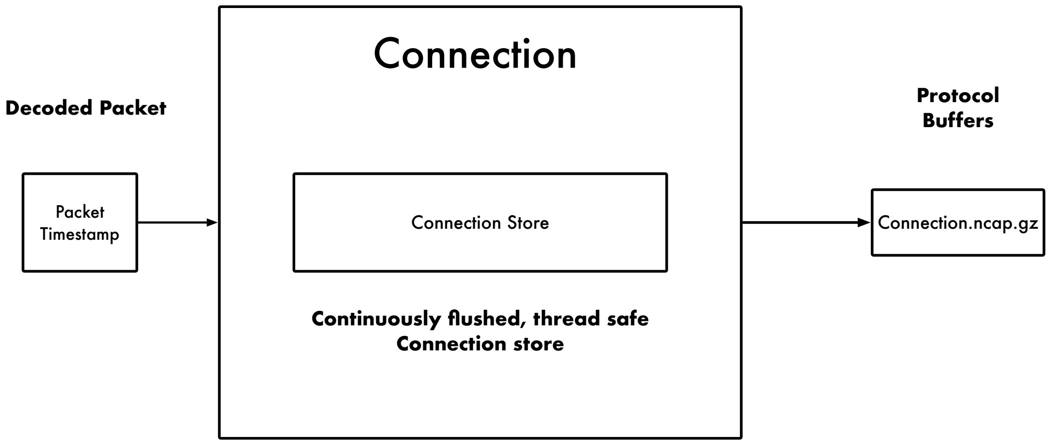


Figure 4.9: Netcap connection encoder

The TLS encoder extracts the *TLS Client Hello* message from the handshake, calculates the *JA3* hash and adds routing information from the underlying TCP packet.

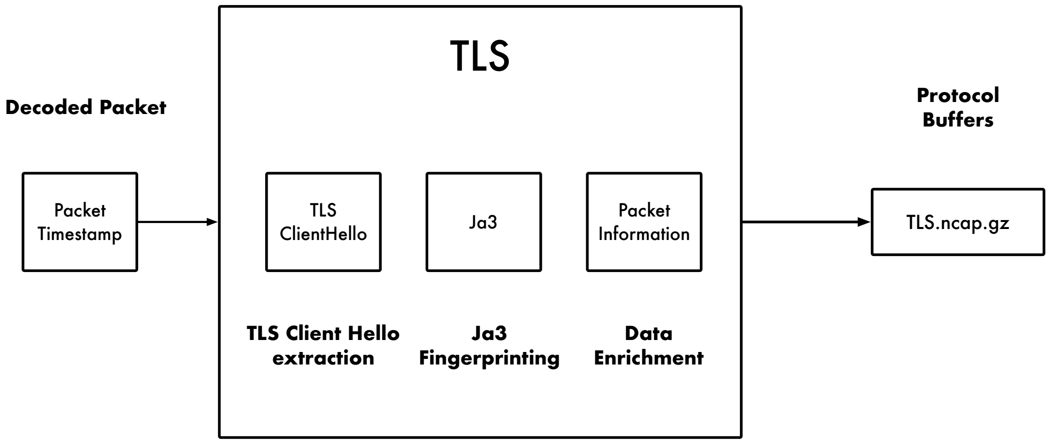


Figure 4.10: Netcap TLS encoder

4.14 Unknown Protocols

Protocols that cannot be decoded will be dumped in the *unknown.pcap* file for later analysis, as this contains potentially interesting traffic that is not represented in the generated output. Separating everything that could not be understood makes it easy to reveal hidden communication channels, which are based on custom protocols.

4.15 Error Log

Errors that happen in the gopacket lib due to malformed packets or implementation errors are written to disk in the *errors.log* file, and can be checked by the analyst later. Each packet that had a decoding error on at least one layer will be added to the *errors.pcap*. An entry to the error log has the following format:

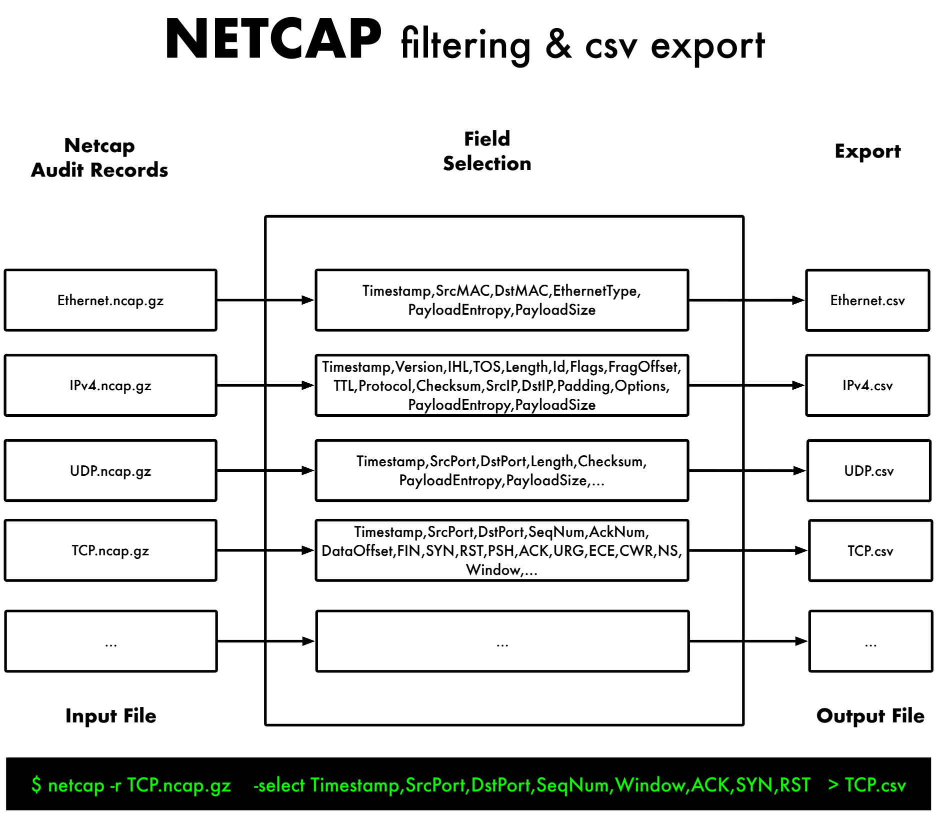
```
<UTC Timestamp>
Error: <Description>
Packet:
<full packet hex dump with layer information>
```

At the end of the error log, a summary of all errors and the number of their occurrences will be appended.

```
...
<error name>: <number of occurrences>
...
```

4.16 Filtering and Export

Netcap offers a simple interface to filter for specific fields and select only those of interest. Filtering and exporting specific fields can be performed with all available audit record types, over a uniform command-line interface. By default, output is generated as CSV with the field names added as first line. It is also possible to use a custom separator string. Fields are exported in the order they are named in the select statement. Sub structures of audit records (for example IPv4Options from an IPv4 packet), are converted to a human readable string representation. More examples for using this feature on the command-line can be found in the usage section.


NETCAP filtering & csv export
Netcap Audit Records
Field Selection
Export
Ethernet.ncap.gz
IPv4.ncap.gz
UDP.ncap.gz
TCP.ncap.gz
...
Timestamp,SrcMAC,DstMAC,EthernetType, PayloadEntropy,PayloadSize
Timestamp,Version,IHL,TOS,Length,Id,Flags,FragOffset, TTL,Protocol,Checksum,SrcIP,DstIP,Padding,Options, PayloadEntropy,PayloadSize
Timestamp,SrcPort,DstPort,Length,Checksum, PayloadEntropy,PayloadSize,...
Timestamp,SrcPort,DstPort,SeqNum,AckNum, DataOffset,FIN,SYN,RST,PSH,ACK,URG,ECE,CWR,NS, Window,...
...
Ethernet.csv
IPv4.csv
UDP.csv
TCP.csv
...
Input File
Output File

```
$ netcap -r TCP.ncap.gz   -select Timestamp,SrcPort,DstPort,SeqNum,Window,ACK,SYN,RST   > TCP.csv
```

Figure 4.11: Netcap filtering and CSV export

4.17 Dataset Labeling

The term *labeling* refers to the procedure of adding classification information to each audit record. For the purpose of intrusion detection this is usually a label stating whether the record is normal or malicious. This is called binary classification, since there are just two choices for the label (good / bad) [BK14]. Efficient and precise creation of labeled datasets is important for supervised machine learning techniques. To create labeled data, *Netcap* parses logs produced by *suricata* and extracts information for each alert. The quality of labels therefore depends on the quality of the used ruleset. In the next step it iterates over the data generated by itself and maps each alert to the corresponding packets, connections or flows. This takes into account all available information on the audit records and alerts. More details on the mapping logic can be found in the implementation chapter. While labeling is usually performed by marking all packets of a known malicious IP address, *Netcap* implements a more granular and thus more precise approach of mapping labels for each record. Labeling happens asynchronously for each audit record type in a separate goroutine.

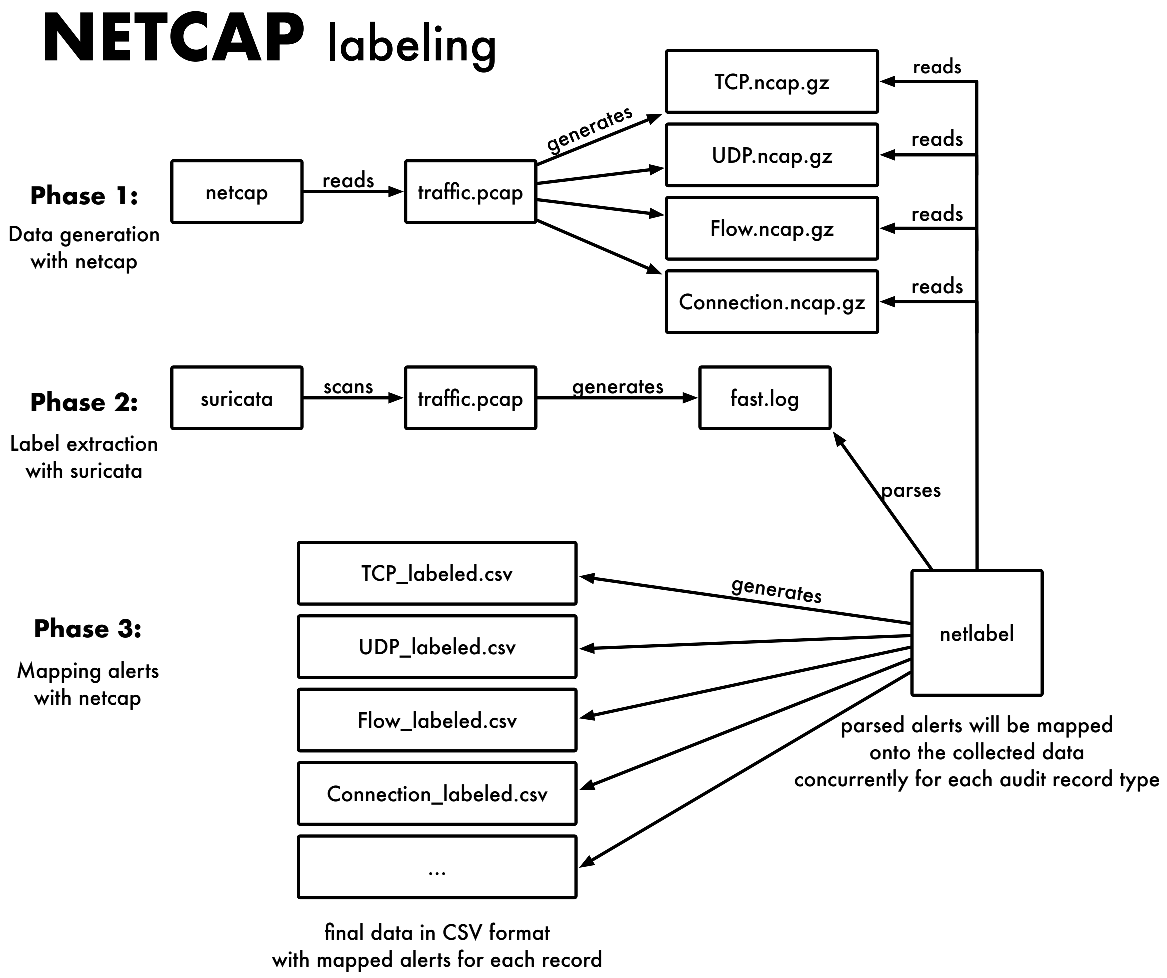


Figure 4.12: Netcap labeling

4.18 Sensors

Using *Netcap* as a data collection mechanism, sensor agents can be deployed to export the traffic they see to a central collection server. This is especially interesting for internet of things (IoT) applications, since these devices are placed inside isolated networks and thus the operator does not have any information about the traffic the device sees. Although Go was not specifically designed for this application, it is an interesting language for embedded systems. Each binary contains the complete runtime, which increases the binary size but requires no installation of dependencies on the device itself. Data exporting currently takes place in batches over UDP sockets. Transferred data is compressed in transit and encrypted with the public key of the collection server. Asymmetric encryption was chosen, to avoid empowering an attacker who compromised a sensor, to decrypt traffic of all sensors communicating with the collection server. To increase the performance, in the future this could be replaced with using a symmetric cipher, together with a solid concept for key rotation and distribution. Sensor agents do not write any data to disk and instead keep it in memory before exporting it.

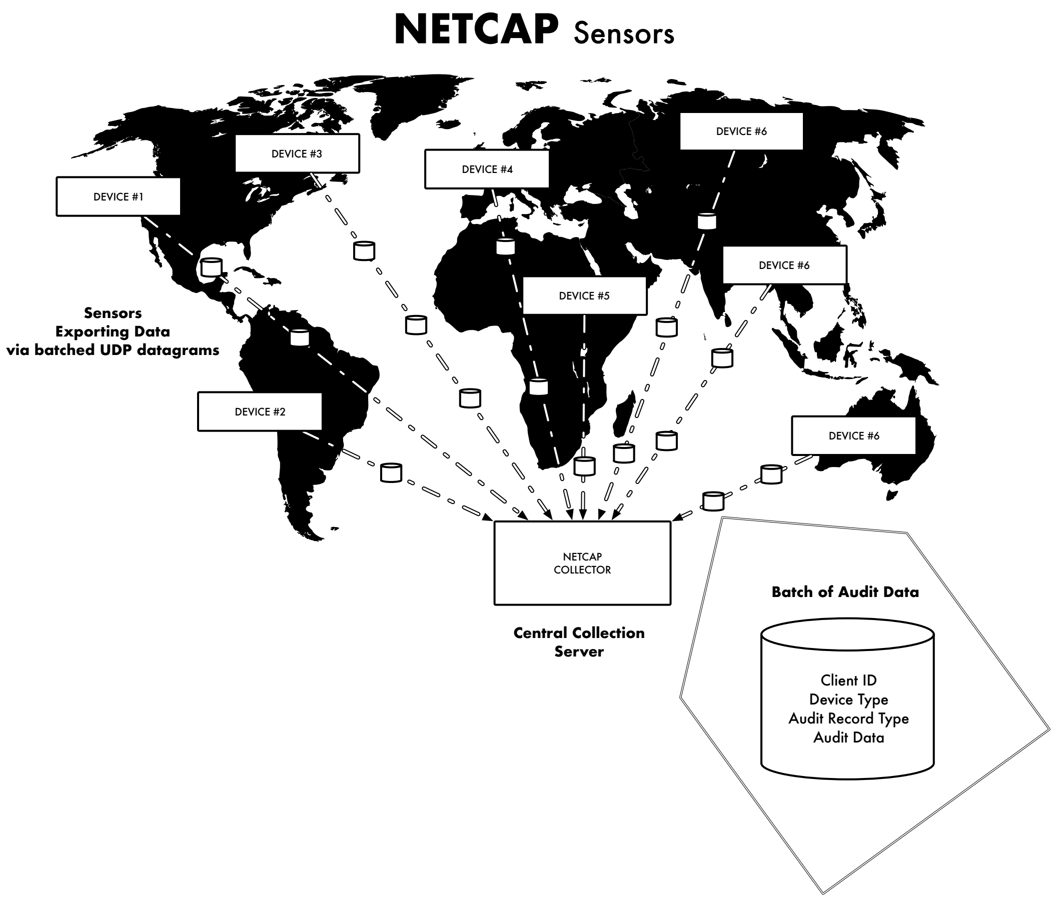


Figure 4.13: Netcap sensors

4.19 Sensor Data Pipe

When sending out batches of audit records, the *Netcap* data pipeline gets slightly modified. After writing data into the atomic writer, it will be transformed into a delimited record and written into a channel. This channel can be used to fetch delimited records from another routine. The sensor sets up this data pipe for each requested audit record type, and then spawns a routine for each of them, that continuously reads from the supplied channel. The data is being read until the total size of bytes would exceed the total size of the buffer. In that case, the current record will be buffered and the remaining data of the batch will be sent to the collector.

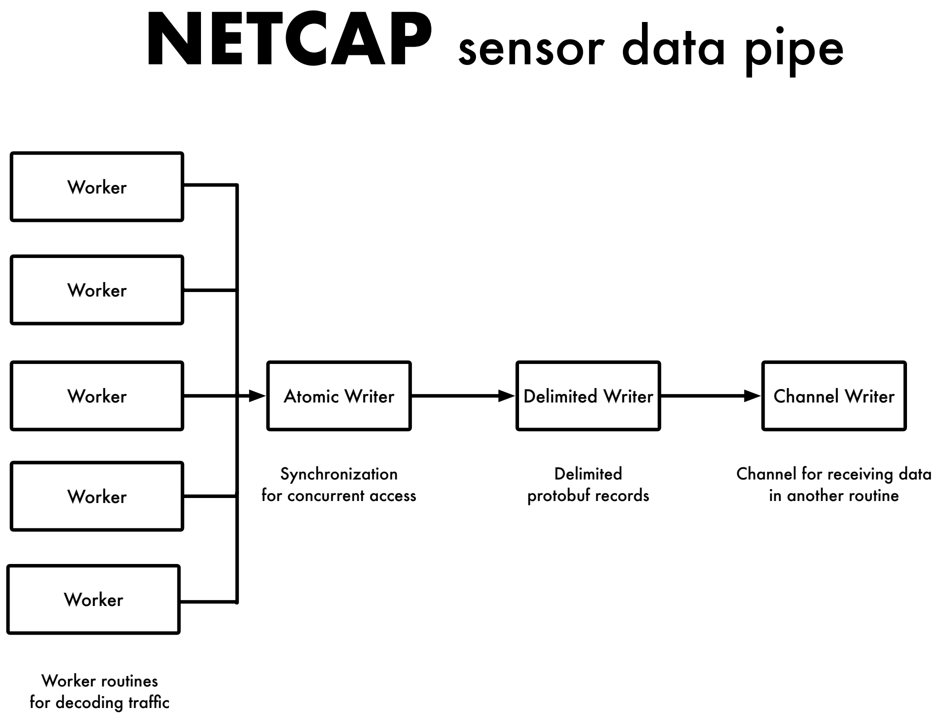

NETCAP sensor data pipe
Worker
Worker
Worker
Worker
Worker
Worker routines for decoding traffic
Atomic Writer
Synchronization for concurrent access
Delimited Writer
Delimited protobuf records
Channel Writer
Channel for receiving data in another routine

Figure 4.14: Netcap sensor pipeline

4.20 Collection Server

The collector node receives data from multiple sensor sources, via batched messages with audit data of a specific type. For now, each received batch of audit data is stored on the file system, separately for each client. For example, if a client has the identifier "xyz", a folder named "xyz" will be created, and inside the different *Netcap* files for each types will be placed. Every time a new batch is received, the data it carries will be appended to the corresponding file on disk. A batch of data is modeled as follows in *netcap.proto*:

```
type Batch struct {
        ClientID            string
        MessageType         string
        Size                int32
        Data                []byte
}
```

Many systems are able to accept protocol buffers as input, alternatively the audit records can be also converted to CSV. In the future, the collection server will be extended to export data to various systems for further analysis and correlation, for example the *elastic stack* or *splunk*.

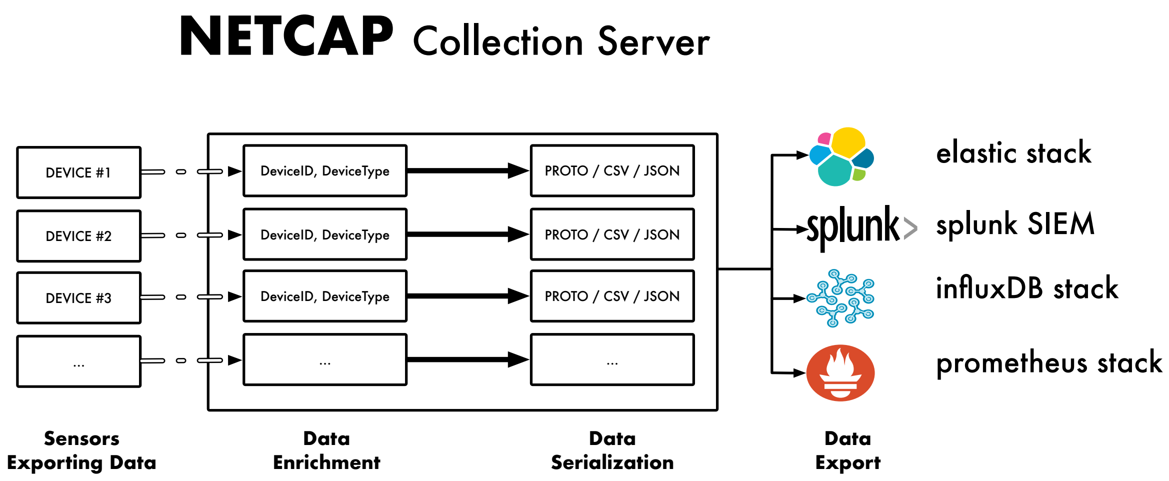


Figure 4.15: Netcap collection server

5 Implementation

This chapter describes internals of the *Netcap* implementation.

5.1 Why Go?

Go, commonly referred to as Golang, is a statically typed and compiled imperative systems programming language released by Google in 2009. It is syntactically similar to the C programming language, but adopted lots of ideas from other languages, such as *Python* and *Erlang*, in order to improve readability and productivity. Commonly used for network programming and backend implementation, Golang is known for its extremely fast compile time and generation of statically linked binaries, which are independent of any libraries on the execution environment. Go is currently available in version 1.11.2 and provides a built-in model for concurrent execution and coordination of asynchronous processes, which was inspired by the Communicating Sequential Processes (CSP) paper. A asynchronous process is called a goroutine, which should not be confused with an operating system (OS) thread. Goroutines are multiplexed onto threads of the OS as required. In case a goroutine blocks, the corresponding OS thread blocks as well, but no other goroutine is affected. Goroutines are less computationally expensive compared to a thread and allocate resources dynamically as needed. For synchronization and messaging, Go offers channels as a lightweight way to communicate between goroutines. This design decision was inspired by the idea of sharing memory by communicating, instead of communicating by sharing memory. Additionally, multi-way concurrent control is provided through select statements, which are used to receive data from channels. Another important aspect of Go is its rich platform and architecture support, which will be described in detail below. Besides that, the language offers a garbage collected runtime, and stack management, in order to combat memory corruptions. However, this should not be confused with the virtual machine approach of the Java runtime. Although Go is not an object oriented programming language, it provides interfaces and the ability to define methods on structures. Go also enforces a uniform programming style for formatting the source code, which greatly helps increasing readability across code from different authors. It is noteworthy that Go provides functionality to circumvent the type system and runtime checks, by reading and writing to arbitrary memory addresses via the *unsafe* package. As of version 0.3.4 *Netcap* does not make use of this package.

5.2 Platform and Architecture Support

Netcap is entirely written in Golang and can be compiled and executed on Mac, Linux and Windows. Supported operating systems (OS), for the current 1.11 version of Go, are: *android*, *darwin*, *dragonfly*, *freebsd*, *linux*, *netbsd*, *openbsd*, *plan9*, *solaris*, *windows*. Golang officially supports the following architectures: *arm64*, *ppc64*, *ppc64le*, *mips64*, *mips64le*, *s390x*, *arm*, *386*, *amd64*. *Netcap* has been developed and tested on *darwin* (macOS mojave

10.14.1, go-1.11.2), but is expected to compile and function seamlessly on all other architectures as well, since it does not make use of any operating or architecture specific features. For creating filepaths, the *filepath* package from the standard library is used. It takes care of using the correct directory separator for the current OS when assembling paths.

5.3 Reading PCAP Files

Support for reading PCAP dump files in Go is provided by the *libpcap* bindings from the *gopacket* library (*github.com/google/gopacket/pcap*). However, this conflicts with the design goals of memory safety, as these bindings rely on the libpcap C library for parsing the input file. Several Golang implementations exist that natively parse PCAP files. After benchmarking and comparing these implementations, it was discovered that all of them make use of the *binary.Read* function from the standard library to unmarshal a byte slice in a predefined structure in memory. Because this function makes use of reflection, it is computationally expensive and cannot provide the desired performance. For *Netcap*s early days, a pure Golang implementation was created that parses the data in a more efficient way. The library and benchmarks have been published on github: *github.com/dreadl0ck/gopcap*. After discovering that a pure go implementation also exists in *gopacket* (*gopacket/pcapgo* package), *Netcap* now uses this as it offers functionality for both reading the PCAP and PCAPNG format, and the option to use a pure go implementation when capturing traffic live on linux. The *pcapgo* package was added to the benchmark comparison, it appears to be only slightly slower than the previously used implementation.

5.4 Reading Traffic from a Network Interface

When reading packets live from an interface, the name of the interface has to be specified with the **-iface** flag. When running on windows or macOS the libpcap bindings must be used for this, if running on linux the native pure go implementation from gopacket is used. In either way, packets are read in a loop from the interface handle with *ZeroCopyReadPacketData() (data []byte, ci gopacket.CaptureInfo, err error)*, until the user cancels with a SIGINT interrupt signal. On receiving the signal a cleanup is performed, all buffers are flushed and written to disk. There are more flags for configuring the live mode: the snapshot length can be adjusted and the network card can be configured to run in promiscous mode, which configures the network adapter to also return packets that were not addressed to itself.

5.5 Concatenating Strings

Concatenating strings can be slow, as joining them with the "+" concatenation operator creates a copy for each operation. Go 1.10 provides a new primitive for efficient string concatenation: *strings.Builder*. A builder behaves like an *io.Writer* and is used thoughout *Netcap* to assemble strings. In the following, it is used to create a human readable representation of an IPv4Option:

```go
func (i IPv4Option) ToString() string {
    var b strings.Builder
    b.WriteString(begin)
    b.WriteString(formatInt32(i.OptionType))
    b.WriteString(sep)
    b.WriteString(formatInt32(i.OptionLength))
    b.WriteString(sep)
    b.WriteString(hex.EncodeToString(i.OptionData))
    b.WriteString(end)
    return b.String()
}
```

Example Output: (134/22/000000020210000200000002000400050006 00ef)

5.6 Atomic Writers

Since writers are shared between multiple concurrent procedures, they are synchronized with a mutual exclusion lock, to allow only one thread at a time to write data. Although synchronization of writers in the *Netcap* data pipe could have also been achieved with channels, mutual exclusion locks were chosen for better performance. Channels have a slight overhead compared to a mutual exclusion lock [goc17]. Synchronization with mutual exclusion locks in Go can be implemented by adding a *Mutex* from the *sync* package, as an anonymous field to a structure. This makes the *Lock()* and *Unlock()* methods available directly on an instance of the structure.

The following shows an example from the collector package, the *AtomicPcapGoWriter* is used for writing packets with unknown layers or decoding errors into the *errors.pcap* file. It counts the number of packets written. Note that for performance reasons, this code does not use a *defer* statement for the unlock operation of the *mutex*.

```go
type atomicPcapGoWriter struct {
    count int64
    w     pcapgo.Writer
    sync.Mutex
}

func (a *atomicPcapGoWriter) WritePacket(ci gopacket.CaptureInfo, data []byte) error {
    // sync
    a.Lock()
    a.count++

    // write data
    err := a.w.WritePacket(ci, data)

    // dont use a defer here for performance
    a.Unlock()
    return err
}
```

5.7 Supported Protocols

This is the list of currently supported protocols, it is expected to grow in the future. Note that this represents the layers as known from the gopacket library, and follows their naming convention.

Name	Layer	Description
Ethernet	Link	IEEE 802.3 Ethernet Protocol
ARP	Link	Address Resolution Procotol
Dot1Q	Link	IEEE 802.1Q, virtual LANs on an Ethernet network
Dot11	Link	IEEE 802.11 Wireless LAN
LinkLayerDiscovery	Link	IEEE 802.1AB Station and Media Access Control Connectivity Discovery
EthernetCTP	Link	diagnostic protocol included in the Xerox Ethernet II specification
EthernetCTPReply	Link	reply to an ethernet ctp packet
LinkLayerDiscoveryInfo	Link	decoded details for a set of LinkLayerDiscoveryValues
LLC	Link	IEEE 802.2 LLC
SNAP	Link	mechanism for multiplexing, on networks using IEEE 802.2 LLC
IPv4	Network	Internet Protocol version 4
IPv6	Network	Internet Protocol version 6
IPv6HopByHop	Network	IPv6 Hop-by-Hop Header
IGMP	Network	Internet Group Management Protocol
ICMPv4	Network	Internet Control Message Protocol v4
ICMPv6	Network	Internet Control Message Protocol v6
ICMPv6NeighborAdvertisement	Network	Neighbor Discovery Protocol
ICMPv6RouterAdvertisement	Network	Neighbor Discovery Protocol
ICMPv6Echo	Network	Neighbor Discovery Protocol
ICMPv6NeighborSolicitation	Network	Neighbor Discovery Protocol
ICMPv6RouterSolicitation	Network	Neighbor Discovery Protocol
UDP	Transport	User Datagram Protocol
TCP	Transport	Transmission Control Protocol
SCTP	Transport	Stream Control Transmission Protocol
DNS	Application	Domain Name System
DHCPv4	Application	Dynamic Host Configuration version 4
DHCPv6	Application	Dynamic Host Configuration version 6
NTP	Application	Network Time Protocol
SIP	Application	Session Initiation Protocol
HTTP	Application	Hypertext Transfer Protocol

Table 5.2: Netcap protocols

5.8 Protocol Sub Structure Types

There are also types for sub structures in protocol fields, currently implemented are:

Name	Description
Dot11QOS	IEEE 802.11 Quality Of Service
Dot11HTControl	IEEE 802.11 HTC information
Dot11HTControlVHT	IEEE 802.11 HTC information
Dot11HTControlHT	IEEE 802.11 HTC information
Dot11HTControlMFB	IEEE 802.11 HTC information
Dot11LinkAdapationControl	IEEE 802.11 HTC information
Dot11ASEL	IEEE 802.11 HTC information
LLDPChassisID	Link Layer Discovery Protocol information
LLDPPortID	Link Layer Discovery Protocol information
LinkLayerDiscoveryValue	Link Layer Discovery Protocol information
LLDPSysCapabilities	Link Layer Discovery Protocol information
LLDPCapabilities	Link Layer Discovery Protocol information
LLDPMgmtAddress	Link Layer Discovery Protocol information
LLDPOrgSpecificTLV	Link Layer Discovery Protocol information
IPv4Option	IPv4 option
ICMPv6Option	ICMPv6 option
TCPOption	TCP option
DNSResourceRecord	Domain Name System resource record
DNSSOA	Domain Name System start of authority record
DNSSRV	Domain Name System service record
DNSMX	Mail exchange record
DNSQuestion	Domain Name System request for a single domain
DHCPOption	DHCP v4 option
DHCPv6Option	DHCP v6 option
IGMPv3GroupRecord	IGMPv3 group records for a membership report
IPv6HopByHopOption	IPv6 hop by hop extension TLV option
IPv6HopByHopOptionAlignment	Hop By Hop Option Alignment

Table 5.4: Netcap protocol sub structures

5.9 Available Fields

5.9.1 Layer Encoders

Layer	NumFields	Fields
TCP	22	Timestamp, SrcPort, DstPort, SeqNum, AckNum, DataOffset, FIN, SYN, RST, PSH, ACK, URG, ECE, CWR, NS, Window, Checksum, Urgent, Padding, Options, PayloadEntropy, PayloadSize
UDP	7	Timestamp, SrcPort, DstPort, Length, Checksum, PayloadEntropy, PayloadSize
IPv4	17	Timestamp, Version, IHL, TOS, Length, Id, Flags, FragOffset, TTL, Protocol, Checksum, SrcIP, DstIP, Padding, Options, PayloadEntropy, PayloadSize
IPv6	12	Timestamp, Version, TrafficClass, FlowLabel, Length, NextHeader, HopLimit, SrcIP, DstIP, PayloadEntropy, PayloadSize, HopByHop
DHCPv4	16	Timestamp, Operation, HardwareType, HardwareLen, HardwareOpts, Xid, Secs, Flags, ClientIP, YourClientIP, NextServerIP, RelayAgentIP, ClientHWAddr, ServerName, File, Options
DHCPv6	7	Timestamp, MsgType, HopCount, LinkAddr, PeerAddr, TransactionID, Options
ICMPv4	5	Timestamp, TypeCode, Checksum, Id, Seq
ICMPv6	3	Timestamp, TypeCode, Checksum
ICMPv6Echo	3	Timestamp, Identifier, SeqNumber
ICMPv6NeighborSolicitation	3	Timestamp, TargetAddress, Options
ICMPv6RouterSolicitation	2	Timestamp, Options
DNS	18	Timestamp, ID, QR, OpCode, AA, TC, RD, RA, Z, ResponseCode, QDCount, ANCount, NSCount, ARCount, Questions, Answers, Authorities, Additionals
ARP	10	Timestamp, AddrType, Protocol, HwAddressSize, ProtAddressSize, Operation, SrcHwAddress, SrcProtAddress, DstHwAddress, DstProtAddress
Ethernet	6	Timestamp, SrcMAC, DstMAC, EthernetType, PayloadEntropy, PayloadSize

Table 5.6: Netcap Layer Encoder Fields Part 1

Layer	NumFields	Fields
Dot1Q	5	Timestamp, Priority, DropEligible, VLANIdentifier, Type
Dot11	14	Timestamp, Type, Proto, Flags, DurationID, Address1, Address2, Address3, Address4, SequenceNumber, FragmentNumber, Checksum, QOS, HTControl
NTP	15	Timestamp, LeapIndicator, Version, Mode, Stratum, Poll, Precision, RootDelay, RootDispersion, ReferenceID, ReferenceTimestamp, OriginTimestamp, ReceiveTimestamp, TransmitTimestamp, ExtensionBytes
SIP	3	Timestamp, OrganizationalCode, Type
IGMP	13	Timestamp, Type, MaxResponseTime, Checksum, GroupAddress, SupressRouterProcessing, RobustnessValue, IntervalTime, SourceAddresses, NumberOfGroupRecords, NumberOfSources, GroupRecords, Version
LLC	6	Timestamp, DSAP, IG, SSAP, CR, Control
IPv6HopByHop	2	Timestamp, Options
SCTP	5	Timestamp, SrcPort, DstPort, VerificationTag, Checksum
SNAP	3	Timestamp, OrganizationalCode, Type
LinkLayerDiscovery	5	Timestamp, ChassisID, PortID, TTL, Values
ICMPv6NeighborAdvertisement	4	Timestamp, Flags, TargetAddress, Options
ICMPv6RouterAdvertisement	7	Timestamp, HopLimit, Flags, RouterLifetime, ReachableTime, RetransTimer, Options
EthernetCTP	2	Timestamp, SkipCount
EthernetCTPReply	4	Timestamp, Function, ReceiptNumber, Data
LinkLayerDiscoveryInfo	8	Timestamp, PortDescription, SysName, SysDescription, SysCapabilities, MgmtAddress, OrgTLVs, Unknown

Table 5.8: Netcap Layer Encoder Fields Part 2

5.9.2 Custom Encoders

Name	NumFields	Fields
TLS	27	Timestamp, Type, Version, MessageLen, HandshakeType, HandshakeLen, HandshakeVersion, Random, SessionIDLen, SessionID, CipherSuiteLen, ExtensionLen, SNI, OSCP, CipherSuites, CompressMethods, SignatureAlgs, SupportedGroups, SupportedPoints, ALPNs, Ja3, SrcIP, DstIP, SrcMAC, DStMAC, SrcPort, DstPort
LinkFlow	9	TimestampFirst, TimestampLast, Proto, SrcMAC, DstMAC, Size, NumPackets, UID, Duration
NetworkFlow	9	TimestampFirst, TimestampLast, Proto, SrcIP, DstIP, Size, NumPackets, UID, Duration
TransportFlow	9	TimestampFirst, TimestampLast, Proto, SrcPort, DstPort, Size, NumPackets, UID, Duration
HTTP	14	Timestamp, Proto, Method, Host, UserAgent, Referer, ReqCookies, ReqContentLength, URL, ResContentLength, ContentType, StatusCode, SrcIP, DstIP
Flow	17	TimestampFirst, LinkProto, NetworkProto, TransportProto, ApplicationProto, SrcMAC, DstMAC, SrcIP, SrcPort, DstIP, DstPort, Size, AppPayloadSize, NumPackets, UID, Duration, TimestampLast
Connection	17	TimestampFirst, LinkProto, NetworkProto, TransportProto, ApplicationProto, SrcMAC, DstMAC, SrcIP, SrcPort, DstIP, DstPort, Size, AppPayloadSize, NumPackets, UID, Duration, TimestampLast

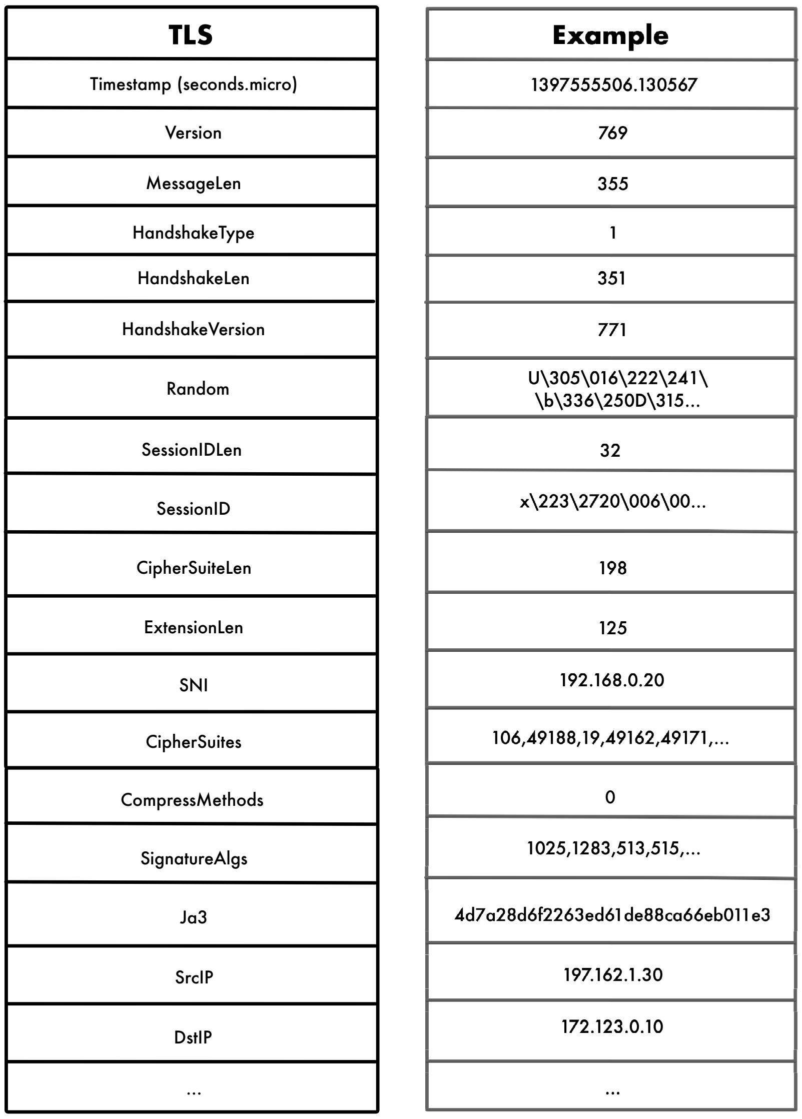
Table 5.10: Netcap Custom Encoder Fields

5.10 TLS Handshakes

In order to be able to identify the initiators of encrypted connections and enhance encrypted telemetry, *Netcap* provides information about handshakes used to create a TLS / SSL secured connection. The *TLS ClientHello* is extracted from TLS v1.1 and v1.2 traffic. Additionally, for each *ClientHello* message the corresponding JA3 hash is calculated and added to the data structure. It will be explained in more detail below. The audit is enriched with context information from the corresponding packet, such as addresses and ports.

```
type TLSClientHello struct {
    Timestamp                string
    Type                     int32
    Version                  int32
    MessageLen               int32
    HandshakeType            int32
    HandshakeLen             uint32
    HandshakeVersion         int32
    Random                   bytes
    SessionIDLen             uint32
    SessionID                bytes
    CipherSuiteLen           int32
    ExtensionLen             int32
    SNI                      string
    OSCP                     bool
    CipherSuites             []int32
    CompressMethods          []int32
    SignatureAlgs            []int32
    SupportedGroups          []int32
    SupportedPoints          []int32
    ALPNs                    []string
    Ja3                      string
    SrcIP                    string
    DstIP                    string
    SrcMAC                   string
    DstMAC                   string
    SrcPort                  int32
    DstPort                  int32
    Extensions               []int32
}
```

The following shows a simple example, note that not all available fields are being displayed and some are truncated for brevity:



TLS	Example
Timestamp (seconds.micro)	1397555506.130567
Version	769
MessageLen	355
HandshakeType	1
HandshakeLen	351
HandshakeVersion	771
Random	U\305\016\222\241\ \b\336\250D\315...
SessionIDLen	32
SessionID	x\223\2720\006\00...
CipherSuiteLen	198
ExtensionLen	125
SNI	192.168.0.20
CipherSuites	106,49188,19,49162,49171,...
CompressMethods	0
SignatureAlgs	1025,1283,513,515,...
Ja3	4d7a28d6f2263ed61de88ca66eb011e3
SrcIP	197.162.1.30
DstIP	172.123.0.10
...	...

Figure 5.1: Netcap TLS data excerpt

5.10.1 JA3 Fingerprinting

Developed by engineers from salesforce and open sourced in July 2017, JA3 is a technique to create a fingerprint of a TLS client hello message, in order to identify the client that is establishing an encrypted connection. For *Netcap*, a pure Go implementation was created (*github.com/dreadl0ck/ja3*). It outperforms the python reference implementation by a factor of 20, and currently finds 30% more handshakes in the test dump file.

JA3 collects the decimal values of the bytes for the following fields in the order they appear, and concatenates them in the named order: **SSL Version, Accepted Ciphers, List of Extensions, Elliptic Curves, Elliptic Curve Formats**.

Those values are then concatenated together sequentially, using "," to delimit a field and "-" to delimit values inside the fields.

Example: 771,49195-49199-52243-52245-49162-57-49171-51-156-53-47-10-255,0-23-35-13-5-13172-18-16-30032-11-10,23-24,0

In case there are no SSL extensions present in the client hello message, the corresponding fields are left empty.

Example: 771,49195-49199-52243-52245-49162-57-49171-51-156-53-47-10-255,,,

The created bare string is then hashed with the MD5 algorithm, to produce a 32 character fingerprint. This hash is the JA3 SSL Client Fingerprint digest. According to the authors, MD5 was chosen for better readability when comparing several fingerprint on a dashboard, despite the risk of hash collisions.

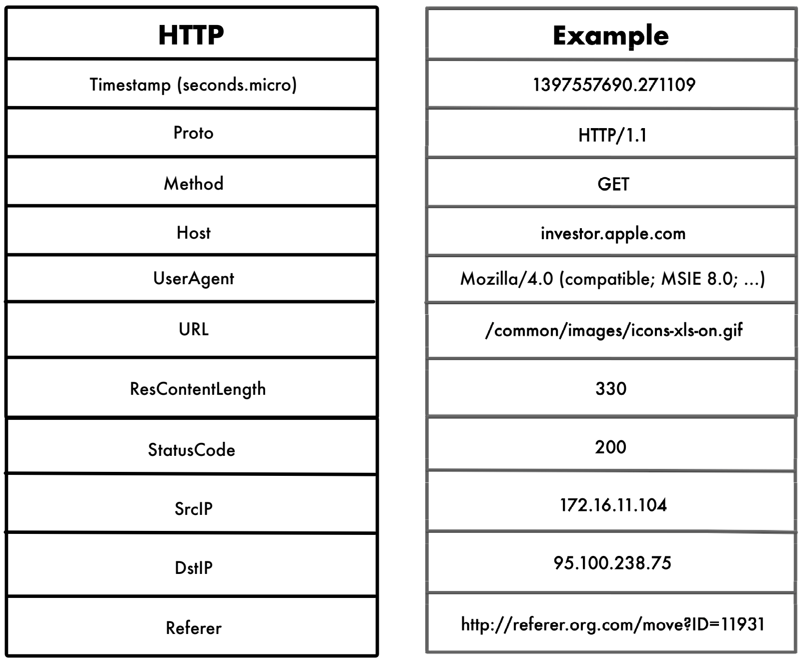
JA3 has several advantages as an indicator of compromise (IoC) compared to IP addresses or domain names. Since JA3 identifies the client application, it is irrelevant if malware uses techniques such as Domain Generation Algorithms (DGA), or several different IP addresses for each command and control host. The idea is to detect malicious communication over SSL based on how the malware communicates, rather than where it connects to.

For heavily restricted environments, where only specific applications are allowed to be installed and used, JA3 is an interesting mechanism for identifying unwanted activity. This can be achieved by creating a whitelist of granted applications and alerting on communication from clients that are not present on the whitelist. The whitelist needs to be updated after changes to client software.

5.11 HTTP

Extracting HTTP audit records has been implemented as a custom encoder. Since a HTTP request or response is often fragmented across several TCP packets, TCP stream reassembly needs to be performed. To accomplish this the *gopacket* implementation for IPv4 stream reassembly (*gopacket/reassembly*) was used, IPv6 stream reassembly is currently not supported. For parsing the HTTP requests and responses, the *net/http* package from the go standard library is used. The *HTTP* audit record type preserves the following information:

```go
type HTTP struct {
    Timestamp         string
    Proto             string
    Method            string
    Host              string
    UserAgent         string
    Referer           string
    ReqCookies        []string
    ReqContentLength  int32
    URL               string
    ResContentLength  int32
    ContentType       string
    StatusCode        int32
    SrcIP             string
    DstIP             string
}
```



HTTP	Example
Timestamp (seconds.micro)	1397557690.271109
Proto	HTTP/1.1
Method	GET
Host	investor.apple.com
UserAgent	Mozilla/4.0 (compatible; MSIE 8.0; ...)
URL	/common/images/icons-xls-on.gif
ResContentLength	330
StatusCode	200
SrcIP	172.16.11.104
DstIP	95.100.238.75
Referer	http://referer.org.com/move?ID=11931

Figure 5.2: Netcap HTTP data excerpt

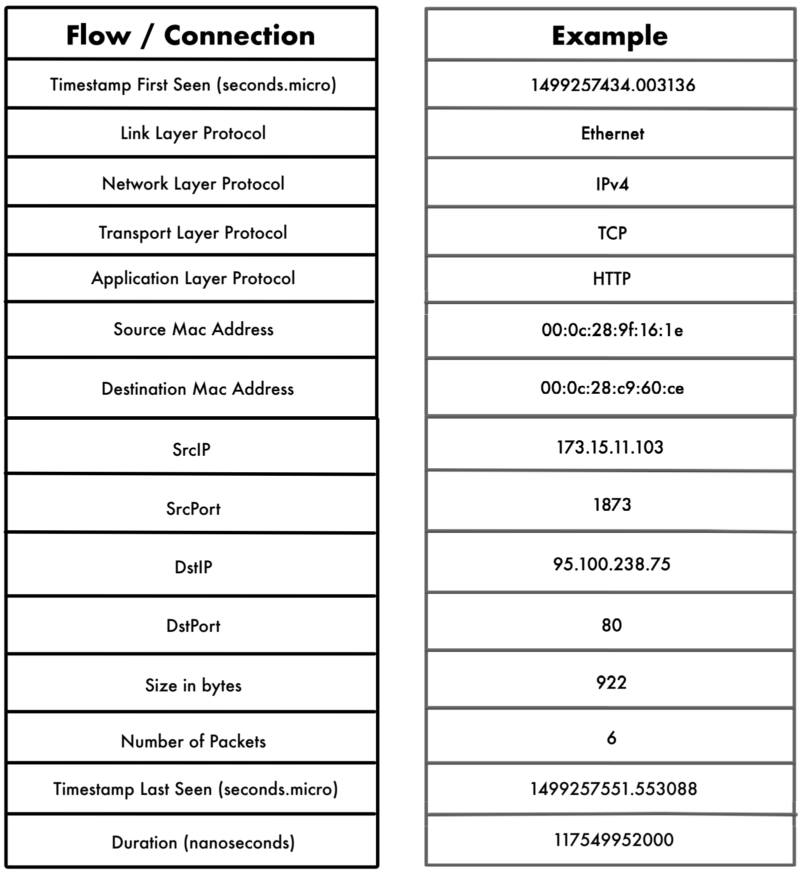
5.12 Flows and Connections

Flow audit records in *Netcap* are unidirectional, whereas *Connections* and *Layer Flows* are bidirectional. *Flows* and *Connections* are continuously flushed, to keep the memory usage as low as possible. Timeout intervals can be configured separately for both types. The first packet seen decides about source and destination of a *Connection* or *Flow.*

Name	Layer	Description
Flow	All	Unidirectional Multi Layer Flow
Connection	All	Bidirectional Multi LayerFlow
LinkFlow	Link	Bidirectional Link Layer Flow
NetworkFlow	Network	Bidirectional Network Layer Flow
TransportFlow	Transport	Bidirectional Transport Layer Flow

Table 5.12: Netcap flows

For Flows and Connections, the following identical information is preserved:



Flow / Connection	Example
Timestamp First Seen (seconds.micro)	1499257434.003136
Link Layer Protocol	Ethernet
Network Layer Protocol	IPv4
Transport Layer Protocol	TCP
Application Layer Protocol	HTTP
Source Mac Address	00:0c:28:9f:16:1e
Destination Mac Address	00:0c:28:c9:60:ce
SrcIP	173.15.11.103
SrcPort	1873
DstIP	95.100.238.75
DstPort	80
Size in bytes	922
Number of Packets	6
Timestamp Last Seen (seconds.micro)	1499257551.553088
Duration (nanoseconds)	117549952000

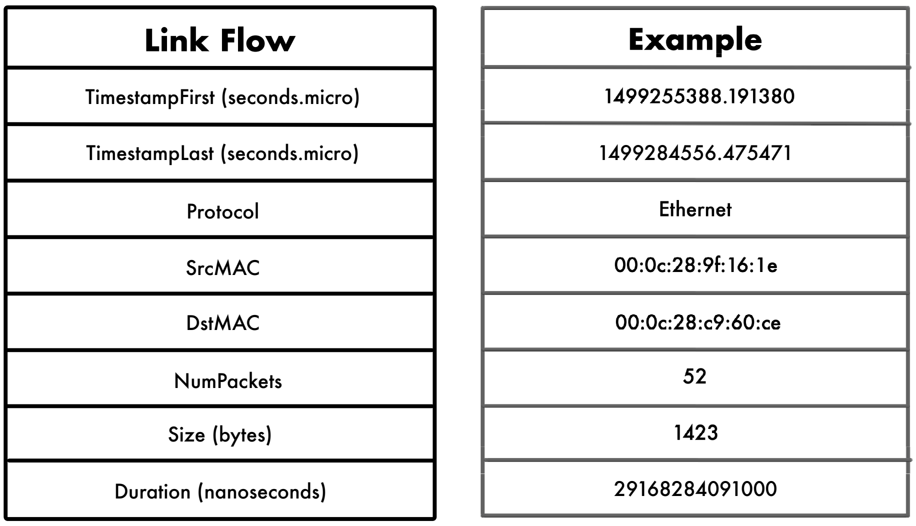
Figure 5.3: Netcap Flows and Connections

5.13 Layer Flows

In order to further decouple *Flows*, they are also exported for each layer, e.g: LinkFlow, NetworkFlow, TransportFlow. As mentioned previously, *Layer Flows* are bidirectional, their UID is the same for both directions. Note that in the following graphics the UID field was omitted.

5.13.1 Link Flow

A *LinkFlow* describes communication on layer 1 of the Internet Protocol Suite. It contains the hardware adresses of the communicating devices, and connection statistics.

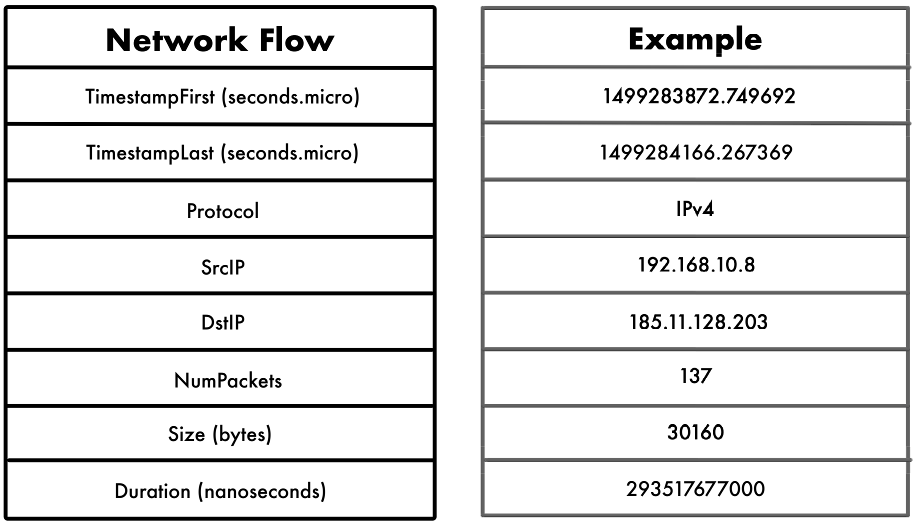


Link Flow	Example
TimestampFirst (seconds.micro)	1499255388.191380
TimestampLast (seconds.micro)	1499284556.475471
Protocol	Ethernet
SrcMAC	00:0c:28:9f:16:1e
DstMAC	00:0c:28:c9:60:ce
NumPackets	52
Size (bytes)	1423
Duration (nanoseconds)	29168284091000

Figure 5.4: Netcap Link Layer Flow example

5.13.2 Network Flow

A *NetworkFlow* describes communication on layer 2 of the Internet Protocol Suite. It contains the ip adresses of the communicating devices, and connection statistics.

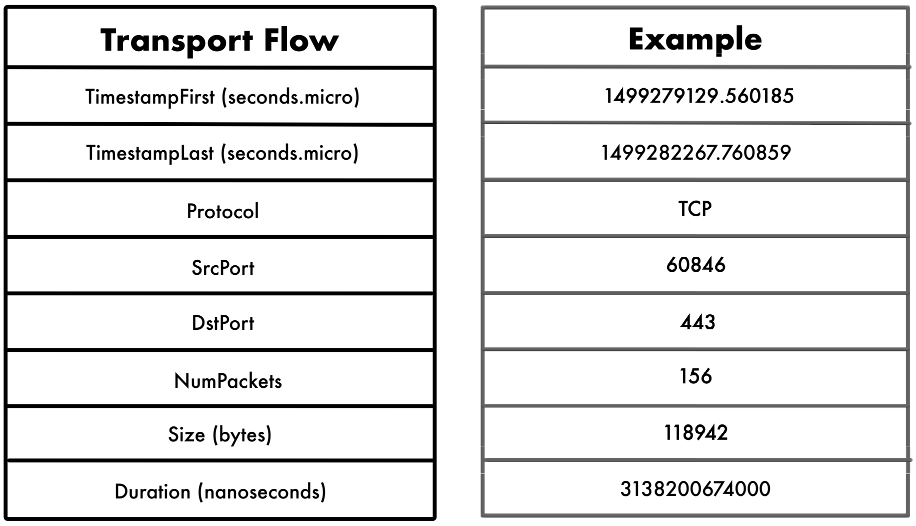


Network Flow	Example
TimestampFirst (seconds.micro)	1499283872.749692
TimestampLast (seconds.micro)	1499284166.267369
Protocol	IPv4
SrcIP	192.168.10.8
DstIP	185.11.128.203
NumPackets	137
Size (bytes)	30160
Duration (nanoseconds)	293517677000

Figure 5.5: Netcap Network Layer Flow example

5.13.3 Transport Flow

A *TransportFlow* describes communication on layer 3 of the Internet Protocol Suite. It contains the ports of the communicating devices, and connection statistics.



Transport Flow	Example
TimestampFirst (seconds.micro)	1499279129.560185
TimestampLast (seconds.micro)	1499282267.760859
Protocol	TCP
SrcPort	60846
DstPort	443
NumPackets	156
Size (bytes)	118942
Duration (nanoseconds)	3138200674000

Figure 5.6: Netcap Transport Layer Flow example

5.14 Sensors & Collection Server

As described in the concept chapter, sensors and the collection server use UDP datagrams for communication. Network communication was implemented using the go standard library. This section will focus on the procedure of encrypting the communication between sensor and collector. For encryption and decryption, cryptographic primitives from the *Golang.org/x/crypto/nacl/box* package are used. The NaCl (pronounced 'Salt') toolkit was developed by the reowned cryptographer Daniel J. Bernstein. The box package uses *Curve25519*, *XSalsa20* and *Poly1305* to encrypt and authenticate messages. It is important to note that the length of messages is not hidden. Netcap uses a thin wrapper around the functionality provided by the nacl package, the wrapper has been published here: *github.com/dreadl0ck/cryptoutils*.

5.14.1 Batch Encryption

The collection server generates a keypair, consisting of two 32 byte (256bit) keys, hex encodes them and writes the keys to disk. The created files are named **pub.key** and **priv.key**. Now, the servers public key can be shared with sensors. Each sensor also needs to generate a keypair, in order to encrypt messages to the collection server with their private key and the public key of the server. To allow the server to decrypt and authenticate the message, the sensor prepends its own public key to each message.

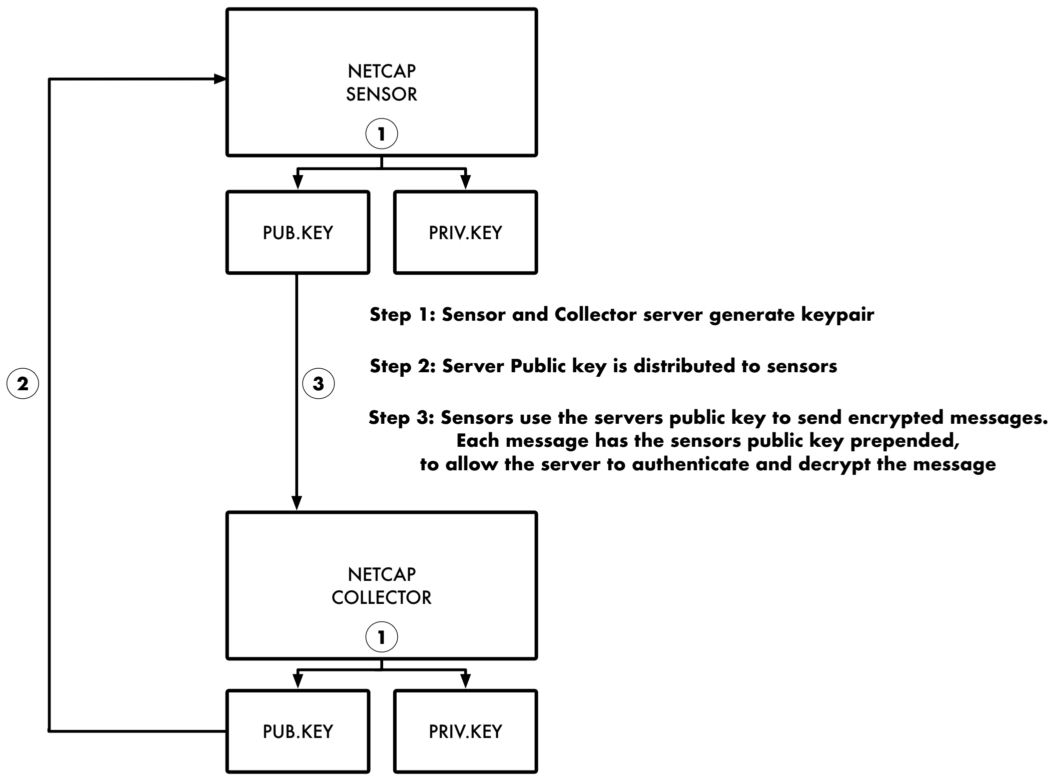

NETCAP SENSOR
1
PUB.KEY
PRIV.KEY
Step 1: Sensor and Collector server generate keypair
Step 2: Server Public key is distributed to sensors
Step 3: Sensors use the servers public key to send encrypted messages. Each message has the sensors public key prepended, to allow the server to authenticate and decrypt the message
2
3
NETCAP COLLECTOR
1
PUB.KEY
PRIV.KEY

Figure 5.7: Netcap collection server

5.14.2 Batch Decryption

When receiving an encrypted batch from a sensor, the server needs to trim off the first 32 bytes, to get the public key of the sensor. Now the message can be decrypted, and decompressed. The resulting bytes are serialized data for a batch protocol buffer. After unmarshalling them into the batch structure, the server can append the serialized audit records carried by the batch, into the corresponding audit record file for the provided client identifier.

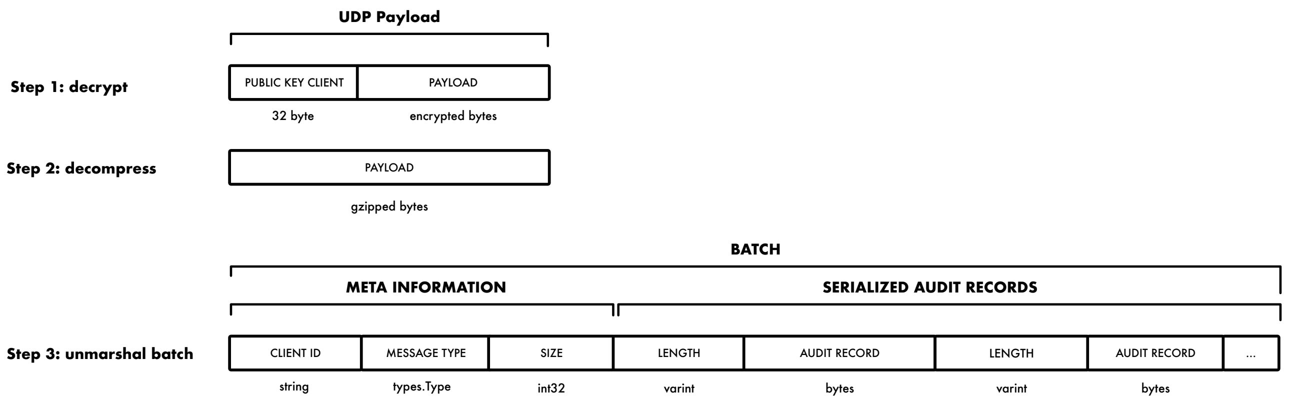


Figure 5.8: Netcap batch decryption

5.15 Packages

5.15.1 Cmd Package

The *cmd* package contains the command-line application. It receives configuration parameters from command-line flags, creates and configures a collector instance, and then starts collecting data from the desired source.

5.15.2 Types Package

The *types* package contains *types.CSV* interface implementations for each supported protocol, to enable converting data to the CSV format. For this purpose, each protocol must provide a *CSVRecord() []string* and a *CSVHeader() []string* function. Additionally, a *NetcapTimestamp() string* function that returns the *Netcap* timestamp must be implemented.

5.15.3 Label Package

The *label* package contains the code for creating labeled datasets. For now, the *suricata* IDS / IPS engine is used to scan the input PCAP and generate alerts. In the future, support could also be added for using YARA. Alerts are then parsed with regular expressions and transformed into the *label.SuricataAlert* type. This could also be replaced by parsing suricatas *eve.json* event logs in upcoming versions. A suricata alert contains the following information:

```
// SuricataAlert is a summary structure of an alerts contents
type SuricataAlert struct {
    Timestamp      string
    Proto          string
    SrcIP          string
    SrcPort        int
    DstIP          string
    DstPort        int
    Classification string
    Description    string
}
```

In the next iteration, the gathered alerts are mapped onto the collected data. For layer types which are not handled separately, this is currently by using solely the timestamp of the packet, since this is the only field required by *Netcap*, however multiple alerts might exist for the same timestamp. To detect this and throw an error, the **-strict** flag can be used. The default is to ignore duplicate alerts for the same timestamp, use the first encountered label and ignore the rest. Another option is to collect all labels that match the timestamp, and append them to the final label with the **-collect** flag. To allow filtering out classifications that shall be excluded, the **-excluded** flag can be used. Alerts matching the excluded classification will then be ignored when collecting the generated alerts. *Flow*, *Connection*, HTTP and TLS records mapping logic also takes source and destination information into consideration. The created output files follow the naming convention: *NetcapType_labeled.csv*. The *label* package includes a standalone command-line application in *label/cmd*.

5.15.4 Sensor Package

The *sensor* package contains the code for a standalone sensor agent, that captures live from an interface and exports the collected data to a central collection server via batched UDP requests.

5.15.5 Server Package

The *server* package implements the UDP collection server, that continuously receives audit data from sensors and stores them on the file system separately for each client.

5.15.6 Utils Package

The *utils* package contains shared utility functions used by several other packages.

5.15.7 Encoder Package

The *encoder* package implements conversion of decoded network protocols to protocol buffers. This has to be defined for each supported protocol. Two types of encoders exist: The *LayerEncoder* and the *CustomEncoder*.

Layer Encoder

A *LayerEncoder* operates on a *gopacket.Layer* and has to provide the *gopacket.LayerType* constant, as well a handler function to receive the layer and the timestamp and convert it into a protocol buffer.

```go
// LayerEncoder contructor
func CreateLayerEncoder(
    lt gopacket.LayerType,
    handler LayerEncoderHandler
) *LayerEncoder

// LayerEncoder instance
type LayerEncoder
    func (d *LayerEncoder) Decode(l gopacket.Layer, timestamp time.Time) error
    func (d *LayerEncoder) Destroy() (name string, size int64)
    func (d *LayerEncoder) GetChan() <-chan []byte
    func (d *LayerEncoder) Init(
        buffer, compress, csv bool,
        out string,
        writeChan bool
    )

// Handler function
type LayerEncoderHandler = func(
    layer gopacket.Layer,
    timestamp string
) proto.Message
```

Example: Ethernet Encoder

The following shows the implementation of the ethernet encoder in *netcap/encoder/eth.go*:

```go
var ethernetEncoder = CreateLayerEncoder(
    layers.LayerTypeEthernet,
    func(layer gopacket.Layer, timestamp string) proto.Message {
        if eth, ok := layer.(*layers.Ethernet); ok {
            return &netcap.Ethernet{
                Timestamp:      timestamp,
                SrcMAC:         eth.SrcMAC.String(),
                DstMAC:         eth.DstMAC.String(),
                EthernetType:   int32(eth.EthernetType),
                PayloadEntropy: Entropy(eth.Payload),
                PayloadSize:    int32(len(eth.Payload)),
            }
        }
        return nil
    },
)
```

Custom Encoder

A *CustomEncoder* operates on a *gopacket.Packet* and is used to decode traffic into abstractions such as *Flows* or *Connections*. To create it a name has to be supplied among three different handler functions to control initialization, decoding and deinitialization. Its handler function receives a *gopacket.Packet* interface type and returns a *proto.Message*. The *postinit* function is called after the initial initialization has taken place, the *deinit* function is used to teardown any additionally created structures for a clean exit. Both functions are optional and can be omitted by supplying *nil* as value.

```go
// CustomEncoder constructor
func CreateCustomEncoder(
    name string,
    postinit func(*CustomEncoder) error,
    handler CustomEncoderHandler,
    deinit func() error
) *CustomEncoder

// CustomEncoder instance
type CustomEncoder
    func (d *CustomEncoder) Decode(info *PacketInfo)
    func (d *CustomEncoder) Destroy() (name string, size int64)
    func (d *CustomEncoder) Init(buffer, compress, csv bool)

// Handler function
type CustomEncoderHandler = func(info *PacketInfo) proto.Message
```

5.15.8 Collector Package

The *collector* package provides an interface for fetching packets from a data source, this can either be a PCAP / PCAPNG file or directly from a named network interface. It is used to implement the command-line interface for *Netcap*. The following shows an overview of the exported API:

```go
func DumpProto(pb proto.Message)
func Entropy(data []byte) (entropy float64)
func GetChan(typ string) <-chan []byte
func NewAtomicPcapGoWriter(w *pcapgo.Writer) *atomicPcapGoWriter
type BatchInfo
type Collector
    func New(config Config) *Collector
    func (c *Collector) CollectBPF(path string, bpf string)
    func (c *Collector) CollectBatch(
        typ string,
        maxSize int,
        bpf string,
        in string,
    ) (*types.Batch, error)
    func (c *Collector) CollectLive(i string, bpf string)
    func (c *Collector) CollectPcap(path string)
    func (c *Collector) CollectPcapNG(path string)
    func (c *Collector) Init()
    func (c *Collector) InitBatching(
        maxSize int,
        bpf string,
        in string,
    ) ([]BatchInfo, *pcap.Handle)
type Config
```

5.16 Unit Tests

Unit tests have been implemented for parts of the core functionality. Currently there are benchmarks for reading pcap and pcapng data, as well as tests and benchmarks for common utility functions, such progress displaying and time conversions. The tests and benchmarks can be executed from the repository root by running:

```
$ go test -v -bench=. ./...
# github.com/dreadl0ck/netcap/collector.test
?       github.com/dreadl0ck/netcap     [no test files]
?       github.com/dreadl0ck/netcap/cmd [no test files]
goos: darwin
goarch: amd64
pkg: github.com/dreadl0ck/netcap/collector
BenchmarkReadPcapNG-12                   1000000    1242 ns/op    1231 B/op    1 allocs/op
BenchmarkReadPcapNGZeroCopy-12           2000000     853 ns/op       0 B/op    0 allocs/op
BenchmarkReadPcap-12                    10000000     185 ns/op      98 B/op    0 allocs/op
PASS
ok      github.com/dreadl0ck/netcap/collector   5.858s
?       github.com/dreadl0ck/netcap/encoder     [no test files]
?       github.com/dreadl0ck/netcap/label       [no test files]
?       github.com/dreadl0ck/netcap/sensor      [no test files]
?       github.com/dreadl0ck/netcap/server      [no test files]
?       github.com/dreadl0ck/netcap/types       [no test files]
=== RUN   TestTimeToString
--- PASS: TestTimeToString (0.00s)
=== RUN   TestStringToTime
```

```
--- PASS: TestStringToTime (0.00s)
goos: darwin
goarch: amd64
pkg: github.com/dreadl0ck/netcap/utils
BenchmarkTimeToStringOld-12            5000000     230 ns/op      64 B/op    4 allocs/op
BenchmarkTimeToString-12              20000000     120 ns/op      80 B/op    3 allocs/op
BenchmarkStringToTime-12              20000000     115 ns/op      32 B/op    1 allocs/op
BenchmarkStringToTimeFieldsFunc-12    10000000     165 ns/op      32 B/op    1 allocs/op
BenchmarkProgressOld-12               30000000    39.5 ns/op       3 B/op    1 allocs/op
BenchmarkProgress-12                  30000000    46.8 ns/op      16 B/op    2 allocs/op
PASS
ok      github.com/dreadl0ck/netcap/utils       10.868s
```

5.17 Data Race Detection Builds

In concurrent programming, shared resources need to be synchronized, in order to guarantee their state when modifying or reading them. If access is not synchronized, race conditions occur, which will lead to faulty program behavior. To avoid this and detect race conditions early in the development cycle, the go toolchain offers compiling the program with the race detector enabled. This will let the application crash with stack traces to assist the developer in debugging, if a data race occurs. Programs with active race detection are slower by the factor of 10 to 100. To compile a Go program with the race detection enabled the **-race** flag must be added to the compilation command.

5.18 Extension

To add support for a new protocol or custom abstraction the following steps need to be performed. First, a type definition of the new audit record type must be added to the *netcap.proto* protocol buffers definitions, as well as a *Type* enumeration following the naming convention with the *NC_* prefix. After recompiling the protocol buffers, a file for the new encoder named after the protocol must be created in the *encoder* package. The new file must contain a variable created with *CreateLayerEncoder* or *CreateCustomEncoder* depending on the desired encoder type. Depending on the choice of the encoder type, the new variable must be added to the *customEncoderSlice* in *encoder/customEncoder.go* or *layerEncoderSlice* in *encoder/layerEncoder.go*. Next, the interface for conversion to CSV must be implemented in the *types* package, by creating a new file with the protocol name and implementing the *CSVHeader() []string*, *CSVRecord() []string*, *NetcapTimestamp() string* functions of the *types.CSV* interface. If the new protocol contains sub-structures, functions to convert them to strings need to be implemented as well. Finally, the *InitRecord(typ types.Type) (record proto.Message)* function in *netcap.go* needs to be updated, to initialize the structure for the new type.

5.19 Caveats

Protocol buffers have a few caveats that developers and researchers should be aware of. First, there are no types for 16 bit signed (*int16*) and unsigned (*uint16*) integers in protobuf, also there is no type for unsigned 8 bit integers (*uint8*). This data type is seen a lot in network protocols, so the question arises how to represent it in protocol buffers. The non-fixed integer types use variable length encoding, so int32 is used instead. The variable-length encoding will take care of not sending the bytes that are not being used. Unfortunately, the mu type is too short for this purpose. Second, protocol buffers require all strings to be encoded as valid UTF-8, otherwise encoding to proto will fail. This means all input data that will be encoded as a string in protobuf must be checked to contain valid UTF-8, or they will create an error upon serialization and end up in the errors.pcap file. If this behavior is not desired strings must be filtered prior to setting them on the protocol buffer instances. Another thing that has to be kept in mind is that *Netcap* processes packets in parallel, thus the order in which packets are written to the dump file is not guaranteed. In experiments, no mixup was detected, and records were tracked in the correct order. However, under heavy load conditions or with a high number of workers, this might be different. Because of this caveat, the *Netcap* specification requires each record to preserve the timestamp, in order to allow sorting the packets afterwards, if required.

5.20 Benchmarks

Processing time depends heavily on the type of traffic, configuration and system performance. Processing times on the development machine (a recent MacBook Pro with 32 GB 2400 MHz DDR4, 2,9 GHz Intel Core i9, running macOS Mojave 10.14), using the default configuration with 4096 byte buffers, compression, all 40+ encoders, 1000 workers and a packet buffer size of 100, can be found in the experiments section in the evaluation chapter.

5.21 Disclaimer

Netcap was developed in a short timeframe as a research project and thus was neither tested nor developed to run in a production environment. The project may contain bugs, that have not yet been discovered. Error handling is not very graceful, in many cases that could have been handled otherwise, the program panics in order to assist in debugging with a stack trace. Until there are further unit tests and the error handling is more robust, using *Netcap* for other purposes than research is not recommended!

5.22 License

Netcap is licensed under the GNU General Public License v3, which is a very permissive open source license, that allows others to do almost anything they want with the project, except to distribute closed source versions. This license type was chosen with *Netcap*s research purpose in mind, and in the hope that it leads to further improvements and new capabilities contributed by other researchers on the long term.

5.23 Code Statistics

The following shows a summary of statistics regarding lines of code (LoC) in *Netcap* version 0.3.3 (excluding generated code for the protocol buffers):

Package	Language	Files	Blank	Comment	Code
collector	Go	13	273	371	882
types	Go	38	232	465	1721
label	Go	15	365	413	1266
encoder	Go	48	559	957	3299
cmd	Go	3	57	64	282
utils	Go	2	39	44	156
netcap.proto	Protobuf	1	93	100	660
total	Go	126	1693	2520	8296

Table 5.14: *Netcap* code statistics for version 0.3.3

The *Netcap* source code contains many descriptive comments (2.5k lines of comments on 8.2k lines of code as of version 0.3.3), in order to ease future development by other researchers and to make implementation details comprehensible.

6 Evaluation

This chapter demonstrates installation and basic usage of the *Netcap* command-line tool and library interface. Afterwards a series of experiments is conducted, in which features from a recent dataset in PCAPNG format will be extracted with *Netcap*, labeled, and used for classification of malicious behavior with Tensorflow and a deep neural network.

6.1 Setup

The repository is available on github (*github.com/dreadl0ck/netcap*).

6.1.1 Installation

Installation via go get:

```
$ go get -u github.com/dreadl0ck/netcap/...
```

To install the command-line tool:

```
$ go build -o $(go env GOPATH)/bin/netcap -i github.com/dreadl0ck/netcap/cmd
```

To cross compile for other architectures, set the *GOARCH* and *GOOS* environment variables. For example to cross compile a binary for *linux amd64*:

```
$ GOARCH=amd64 GOOS=linux go build -o netcap -i github.com/dreadl0ck/netcap/cmd
```

6.1.2 Buildsystem

Netcap uses the *zeus* buildsystem, it can be found on github along with installation instructions: *github.com/dreadl0ck/zeus*. However, the project can easily be installed without *zeus*. All shell scripts needed for installation can be found in the *zeus/generated* directory as standalone versions.

To install the *Netcap* command-line tool and the library with *zeus*, run:

```
$ zeus install
```

6.2 Netcap command-line Tool

In the following common *Netcap* operations on the command-line are presented and explained. If output was truncated for brevity, it will be denoted by three dots.

6.2.1 Help

The **-h** flag can be used to print an overview of all available command-line flags and their default values to the terminal:

```
Usage of netcap:
-allowmissinginit
        support streams without SYN/SYN+ACK/ACK sequence
-assembly_debug_log
        If true, the github.com/google/gopacket/reassembly library
        will log verbose debugging information (at least one line per packet)
-assembly_memuse_log
        If true, the github.com/google/gopacket/reassembly library
        will log information regarding its memory use every once in a while.
-bpf string
        supply a BPF filter to use prior to processing packets with netcap
-buf
        buffer data in memory before writing to disk (default true)
-check
        check number of occurences of the separator, in fields of an audit record file
-checksum
        check TCP checksum
-comp
        compress output with gzip (default true)
-conn-flush-interval int
        flush connections every X flows (default 10000)
-conn-timeout int
        close connections older than X seconds (default 60)
-cpuprof
        create cpu profile
-csv
        print output data as csv with header line
-debug
        display debug information
-dump
        dump HTTP request/response as hex
-encoders
        show all available encoders
-exclude string
        exclude specific encoders
-fields
        print available fields for an audit record file and exit
-files string
        path to create file for HTTP 200 OK responses
-flow-flush-interval int
        flush flows every X flows (default 2000)
-flow-timeout int
        close flows older than X seconds (default 30)
-flushevery int
        flush assembler every N packets (default 10000)
-header
        print audit record file header and exit
-iface string
        attach to network interface and capture in live mode
-ignore-unknown
        disable writing unknown packets into a pcap file
-ignorefsmerr
        ignore TCP FSM errors
```

```
-include string
        include specific encoders
-memprof
        create memory profile
-memprofile string
        write memory profile
-nodefrag
        if true, do not do IPv4 defrag
-nohttp
        disable HTTP parsing
-nooptcheck
        do not check TCP options (useful to ignore MSS on captures with TSO)
-out string
        specify output directory, will be created if it does not exist
-overview
        print a list of all available encoders and fields
-pbuf int
        set packet buffer size, for channels that feed data to workers (default 100)
-promisc
        toggle promiscous mode for live capture (default true)
-quiet
        be quiet regarding errors (default true)
-r string
        read specified file, can either be a pcap or netcap audit record file
-select string
        select specific fields of an audit records when generating csv or tables
-sep string
        set separator string for csv output (default ",")
-snaplen int
        configure snaplen for live capture from interface (default 1024)
-struc
        print output as structured objects
-table
        print output as table view (thanks @evilsocket)
-tcp-close-timeout int
        close tcp streams if older than X seconds
        (set to 0 to keep long lived streams alive) (default 180)
-tcp-timeout int
        close streams waiting for packets older than X seconds (default 120)
-ts2utc string
        util to convert sencods.microseconds timestamp to UTC
-tsv
        print output as tab separated values
-utc
        print timestamps as UTC when using select csv
-verbose
        be verbose
-version
        print netcap package version and exit
-workers int
        number of workers (default 1000)
-writeincomplete
        write incomplete response
```

6.2.2 Show Audit Record File Header

To display the header of the supplied audit record file, the **-header** flag can be used:

```
$ netcap -r TCP.ncap.gz -header

+----------+--------------------------------------+
|  Field   |                Value                 |
+----------+--------------------------------------+
| Created  | 2018-11-15 04:42:22.411785 +0000 UTC |
| Source   | Wednesday-WorkingHours.pcap          |
| Version  | v0.3.3                               |
| Type     | NC_TCP                               |
+----------+--------------------------------------+
```

6.2.3 Print Structured Audit Records

Audit records can be printed structured, this makes use of the *proto.MarshalTextString()* function. This is sometimes useful for debugging, but very verbose.

```
$ netcap -r TCP.ncap.gz -struc
...
NC_TCP
Timestamp: "1499255023.848884"
SrcPort: 80
DstPort: 49472
SeqNum: 1959843981
AckNum: 3666268230
DataOffset: 5
ACK: true
Window: 1025
Checksum: 2348
PayloadEntropy: 7.836586993143013
PayloadSize: 1460
...
```

6.2.4 Print as CSV

This is the default behavior. First line contains all field names.

```
$ netcap -r TCP.ncap.gz
Timestamp,SrcPort,DstPort,SeqNum,AckNum,DataOffset,FIN,SYN,RST,PSH,ACK,URG,...
1499254962.234259,443,49461,1185870107,2940396492,5,false,false,false,true,true,false,...
1499254962.282063,49461,443,2940396492,1185870976,5,false,false,false,false,true,false,...
...
```

6.2.5 Print as Tab Separated Values

To use a tab as separator, the **-tsv** flag can be supplied:

```
$ netcap -r TCP.ncap.gz -tsv
Timestamp               SrcPort DstPort Length  Checksum PayloadEntropy  PayloadSize
1499254962.084372       49792   1900    145     34831    5.19616448      137
1499254962.084377       49792   1900    145     34831    5.19616448      137
1499254962.084378       49792   1900    145     34831    5.19616448      137
1499254962.084379       49792   1900    145     34831    5.19616448      137
...
```

6.2.6 Print as Table

The **-table** flag can be used to print output as a table. Every 100 entries the table is printed to stdout.

```
$ netcap -r UDP.ncap.gz -table -select Timestamp,SrcPort,DstPort,Length,Checksum
+--------------------+----------+----------+---------+-----------+
|     Timestamp      | SrcPort  | DstPort  | Length  | Checksum  |
+--------------------+----------+----------+---------+-----------+
| 1499255691.722212  | 62109    | 53       | 43      | 38025     |
| 1499255691.722216  | 62109    | 53       | 43      | 38025     |
| 1499255691.722363  | 53       | 62109    | 59      | 37492     |
| 1499255691.722366  | 53       | 62109    | 59      | 37492     |
| 1499255691.723146  | 56977    | 53       | 43      | 7337      |
| 1499255691.723149  | 56977    | 53       | 43      | 7337      |
| 1499255691.723283  | 53       | 56977    | 59      | 6804      |
| 1499255691.723286  | 53       | 56977    | 59      | 6804      |
| 1499255691.723531  | 63427    | 53       | 43      | 17441     |
| 1499255691.723534  | 63427    | 53       | 43      | 17441     |
| 1499255691.723682  | 53       | 63427    | 87      | 14671     |
...
```

6.2.7 Print with Custom Separator

Output can also be generated with a custom separator:

```
$ netcap -r TCP.ncap.gz -sep ";"
Timestamp;SrcPort;DstPort;Length;Checksum;PayloadEntropy;PayloadSize
1499254962.084372;49792;1900;145;34831;5.19616448;137
1499254962.084377;49792;1900;145;34831;5.19616448;137
1499254962.084378;49792;1900;145;34831;5.19616448;137
...
```

6.2.8 Validate generated Output

To ensure values in the generated CSV would not contain the separator string, the **-check** flag can be used. This will determine the expected number of separators for the audit record type, and print all lines to stdout that do not have the expected number of separator symbols. The separator symbol will be colored red with ansi escape secquences and each line is followed by the number of separators in red color. The **-sep** flag can be used to specify a custom separator.

```
$ netcap -r TCP.ncap.gz -check
$ netcap -r TCP.ncap.gz -check -sep=";"
```

6.2.9 Filtering and Export

Netcap offers a simple command-line interface to select fields of interest from the gathered audit records.

Example: Filtering UDP audit records

Show available header fields:

```
$ netcap -r UDP.ncap.gz -fields
Timestamp,SrcPort,DstPort,Length,Checksum,PayloadEntropy,PayloadSize
```

Print all fields for the supplied audit record:

```
$ netcap -r UDP.ncap.gz
1331904607.100000,53,42665,120,41265,4.863994469989251,112
1331904607.100000,42665,53,53,1764,4.0625550894074385,45
1331904607.290000,51190,53,39,22601,3.1861758166070766,31
1331904607.290000,56434,53,39,37381,3.290856864924384,31
1331904607.330000,137,137,58,64220,3.0267194361875682,50
...
```

Selecting fields will also define their order:

```
$ netcap -r UDP.ncap.gz -select Length,SrcPort,DstPort,Timestamp
Length,SrcPort,DstPort,Timestamp
145,49792,1900,1499254962.084372
145,49792,1900,1499254962.084377
145,49792,1900,1499254962.084378
145,49792,1900,1499254962.084379
145,49792,1900,1499254962.084380
...
```

Print selection in the supplied order and convert timestamps to UTC time:

```
$ netcap -r UDP.ncap.gz -select Timestamp,SrcPort,DstPort,Length -utc
2012-03-16 13:30:07.1 +0000 UTC,53,42665,120
2012-03-16 13:30:07.1 +0000 UTC,42665,53,53
2012-03-16 13:30:07.29 +0000 UTC,51190,53,39
2012-03-16 13:30:07.29 +0000 UTC,56434,53,39
2012-03-16 13:30:07.33 +0000 UTC,137,137,58
...
```

To save the output into a new file, simply redirect the standard output:

```
$ netcap -r UDP.ncap.gz -select Timestamp,SrcPort,DstPort,Length -utc > UDP.csv
```

6.2.10 Inclusion & Exclusion of Encoders

The **-encoders** flag can be used to list all available encoders. In case not all of them are desired, selective inlcusion and exclusion is possible, by using the **-include** and **-exclude** flags.

List all encoders:

```
$ netcap -encoders
custom:
+ TLS
+ LinkFlow
+ NetworkFlow
+ TransportFlow
+ HTTP
+ Flow
+ Connection
layer:
+ TCP
+ UDP
+ IPv4
+ IPv6
+ DHCPv4
+ DHCPv6
+ ICMPv4
+ ICMPv6
+ ICMPv6Echo
...
```

Include specific encoders (only those named will be used):

```
$ netcap -r traffic.pcap -include Ethernet,Dot1Q,IPv4,IPv6,TCP,UDP,DNS
```

Exclude encoders (this will prevent decoding of layers encapsulated by the excluded ones):

```
$ netcap -r traffic.pcap -exclude TCP,UDP
```

6.2.11 Applying Berkeley Packet Filters

Netcap will decode all traffic it is exposed to, therefore it might be desired to set a berkeley packet filter, to reduce the workload imposed on *Netcap*. This is possible for both live and offline operation. In case a BPF should be set for offline use, the *gopacket/pcap* package with bindings to the *libpcap* will be used, since setting BPF filters is not yet supported by the native *pcapgo* package.

When capturing live from an interface:

```
$ netcap -iface en0 -bpf "host 192.168.1.1"
```

When reading offline dump files:

```
$ netcap -r traffic.pcap -bpf "host 192.168.1.1"
```

6.2.12 Usage Examples

Read traffic live from interface, stop with *Ctrl-C* (SIGINT):

```
$ netcap -iface eth0
```

Read traffic from a dump file (supports PCAP or PCAPNG):

```
$ netcap -r traffic.pcap
```

Read a netcap dumpfile and print to stdout as CSV:

```
$ netcap -r TCP.ncap.gz
```

Show the available fields for a specific *Netcap* dump file:

```
$ netcap -fields -r TCP.ncap.gz
```

Print only selected fields and output as CSV:

```
$ netcap -r TCP.ncap.gz -select Timestamp,SrcPort,DstPort
```

Save CSV output to file:

```
$ netcap -r TCP.ncap.gz -select Timestamp,SrcPort,DstPort > tcp.csv
```

Print output separated with tabs:

```
$ netcap -r TPC.ncap.gz -tsv
```

Run with 24 workers and disable gzip compression and buffering:

```
$ netcap -workers 24 -buf false -comp false -r traffic.pcapng
```

Parse pcap and write all data to output directory (will be created if it does not exist):

```
$ netcap -r traffic.pcap -out traffic_ncap
```

Convert timestamps to UTC:

```
$ netcap -r TCP.ncap.gz -select Timestamp,SrcPort,Dstport -utc
```

6.2.13 Example Output

Running *Netcap* with a PCAP dumpfile prints the following output to the terminals stdout:

```
$ netcap -r netdump.pcap
                       / |
 _______    ______   _10 |_     _______   ______    ______
/     / \  /    / \ / 01/  |   /     / | /    / \  /    / \
0010100 /|/011010 /|101010/   /0101010/  001010  |/100110  |
01 |  00 |00    00 |  10 | __ 00 |       /    10 |00 |  01 |
10 |  01 |01001010/   00 |/  |01 \_____ /0101000 |00 |__10/|
10 |  00 |00/    / |  10  00/ 00/    / |00    00 |00/   00/
00/   10/  0101000/    0010/   0010010/  0010100/ 1010100/
                                                  00 |
Network Protocol Analysis Toolkit                 00 |
created by Philipp Mieden, 2018                   00/
v0.3.3

+---------------+--------+
|   Setting     | Value  |
+---------------+--------+
| Workers       | 100    |
| MemBuffer     | true   |
| Compression   | true   |
| PacketBuffer  | 100    |
+---------------+--------+

opening netdump.pcap | size: 206 MB
counting packets... done. 256449 packets found in 237.263746ms
initialized 29 layer encoders | buffer size: 4096
initialized 7 custom encoders | buffer size: 4096
done.

Processed 256449 packets (202434809 bytes) in 14.412113465s (errors: 114785, type:4)
Final flush: 43 closed

TCP stats:

+-----------------------+------------+
|      Description      |   Value    |
+-----------------------+------------+
| IPdefrag              | 0          |
| missed bytes          | 96245216   |
| total packets         | 67640      |
| rejected FSM          | 462        |
| rejected Options      | 114076     |
| reassembled bytes     | 39126253   |
| total TCP bytes       | 188067731  |
| conn rejected FSM     | 135        |
| reassembled chunks    | 153        |
| out-of-order packets  | 5844       |
| out-of-order bytes    | 2813069    |
| biggest-chunk packets | 1740       |
| biggest-chunk bytes   | 65145      |
| overlap packets       | 44         |
| overlap bytes         | 6746       |
+-----------------------+------------+

Errors: 114785

+--------------------+---------+
|       Error        | Count   |
+--------------------+---------+
| OptionChecker      | 114076  |
| FSM                | 462     |
| HTTP-response-body  | 69      |
| HTTP-response       | 178     |
+---------------------+---------+

flushed 0 http events. requests 1880 responses 1793

+-----------+-------------+-------------+
| Protocol  | NumPackets  |   Share     |
+-----------+-------------+-------------+
| Payload   | 175840      | 68.56724%   |
| UDP       | 1014        | 0.39540%    |
| ARP       | 383         | 0.14935%    |
| DHCPv4    | 2           | 0.00078%    |
| IGMP      | 2           | 0.00078%    |
| DNS       | 837         | 0.32638%    |
| TCP       | 255044      | 99.45213%   |
| ICMPv4    | 6           | 0.00234%    |
| Ethernet  | 256449      | 100.00000%  |
| IPv4      | 256066      | 99.85065%   |
+-----------+-------------+-------------+

-> total bytes of data written to disk: 13 MB

Active Custom Encoders:
+ TLS
+ LinkFlow
+ NetworkFlow
+ TransportFlow
+ HTTP
+ Flow
+ Connection

done in 14.445120991s
```

Error log example (file errors.log):

```
2012-03-16 13:51:04.16 +0100 CET
Error: Layer type not currently supported
Packet:
-- FULL PACKET DATA (78 bytes) ------------------------------------
00000000  01 00 5e 00 00 0a 00 16  47 9d f2 cd 81 00 c0 d6  |..^.....G.......|
00000010  08 00 45 c0 00 3c 00 00  00 00 02 58 40 f6 c0 a8  |..E..<.....X@...|
00000020  d6 01 e0 00 00 0a 02 05  ec 6c 00 00 00 00 00 00  |.........l......|
00000030  00 00 00 00 00 00 00 00  00 64 00 01 00 0c 01 00  |.........d......|
00000040  01 00 00 00 00 0f 00 04  00 08 0c 02 03 00        |..............|
--- Layer 1 ---
Ethernet        {
    Contents=[..14..]
    Payload=[..64..]
    SrcMAC=00:16:47:9d:f2:cd
    DstMAC=01:00:5e:00:00:0a
    EthernetType=Dot1Q
    Length=0
}
00000000  01 00 5e 00 00 0a 00 16  47 9d f2 cd 81 00        |..^.....G.....|
--- Layer 2 ---
Dot1Q   {
    Contents=[192, 214, 8, 0]
    Payload=[..60..]
    Priority=6
    DropEligible=false
    VLANIdentifier=214
    Type=IPv4
}
00000000  c0 d6 08 00                                        |....|
--- Layer 3 ---
IPv4    {
    Contents=[..20..]
    Payload=[..40..]
    Version=4
    IHL=5
    TOS=192
    Length=60
    Id=0
    Flags=
    FragOffset=0
    TTL=2
    Protocol=UnknownIPProtocol
    Checksum=16630
    SrcIP=192.168.214.1
    DstIP=224.0.0.10
    Options=[]
    Padding=[]
}
00000000  45 c0 00 3c 00 00 00 00  02 58 40 f6 c0 a8 d6 01  |E..<.....X@.....|
00000010  e0 00 00 0a                                        |....|
--- Layer 4 ---
DecodeFailure   Packet decoding error: Layer type not currently supported
00000000  02 05 ec 6c 00 00 00 00  00 00 00 00 00 00 00 00  |...l............|
00000010  00 00 00 64 00 01 00 0c  01 00 01 00 00 00 00 0f  |...d............|
00000020  00 04 00 08 0c 02 03 00
```

6.3 Netlabel command-line Tool

In the following common operations with the *netlabel* tool on the command-line are presented and explained.

6.3.1 Help

To display the available command-line flags, the **-h** flag must be used:

```
$ netlabel -h
Usage of netlabel:
    -collect
        append classifications from alert with duplicate timestamps
        to the generated label
    -description
        use attack description instead of classification for labels
    -disable-layers
        do not map layer types by timestamp
    -exclude string
        specify a comma separated list of suricata classifications
        that shall be excluded from the generated labeled csv
    -out string
        specify output directory, will be created if it does not exist
    -progress
        use progress bars
    -r string
        read specified file, can either be a pcap or netcap audit record file
    -sep string
        set separator string for csv output (default ",")
    -strict
        fail when there is more than one alert for the same timestamp
```

6.3.2 Usage Examples

Scan input pcap and create labeled csv files by mapping audit records in the current directory:

```
$ netlabel -r traffic.pcap
```

Scan input pcap and create output files by mapping audit records from the output directory:

```
$ netlabel -r traffic.pcap -out output_dir
```

Abort if there is more than one alert for the same timestamp:

```
$ netlabel -r taffic.pcap -strict
```

Display progress bar while processing input (experimental):

```
$ netlabel -r taffic.pcap -progress
```

Append classifications for duplicate labels:

```
$ netlabel -r taffic.pcap -collect
```

6.4 Sensors & Collection Server

Both sensor and client can be configured by using the **-addr** flag to specify an ip address and port. To generate a keypair for the server, the **-gen-keypair** flag must be used:

```
$ netcap-server -gen-keypair
wrote keys
$ ls
priv.key    pub.key
```

Now, the server can be started, the location of the file containing the private key must be supplied:

```
$ netcap-server -privkey priv.key -addr 127.0.0.1:4200
```

The server will now be listening for incoming messages. Next, the sensor must be configured. The keypair for the sensor will be generated on startup, but the public key of the server must be provided:

```
$ netcap-sensor -pubkey pub.key -addr 127.0.0.1:4200
got 126 bytes of type NC_ICMPv6RouterAdvertisement expected [126]
got size [73] for type NC_Ethernet
got 73 bytes of type NC_Ethernet expected [73]
got size [27] for type NC_ICMPv6
got size [126] for type NC_ICMPv6RouterAdvertisement
got 126 bytes of type NC_ICMPv6RouterAdvertisement expected [126]
got size [75] for type NC_IPv6
got 75 bytes of type NC_IPv6 expected [75]
got 27 bytes of type NC_ICMPv6 expected [27]
```

The client will now collect the traffic live from the specified interface, and send it to the configured server, once a batch for an audit record type is complete. The server will log all received messages:

```
$ netcap-server -privkey priv.key -addr 127.0.0.1:4200
packet-received: bytes=2412 from=127.0.0.1:57368
decoded batch NC_Ethernet from client xyz
new file xyz/Ethernet.ncap
packet-received: bytes=2701 from=127.0.0.1:65050
decoded batch NC_IPv4 from client xyz
new file xyz/IPv4.ncap
...
```

When stopping the server with a *SIGINT*, all audit record file handles will be flushed and closed properly.

6.5 Library Usage

6.5.1 Golang Library

The *Netcap* Go library (*github.com/dreadl0ck/netcap*) provides support for capturing *Netcap* audit records either live or offline, and for reading back data from disk.

```go
func Count(filename string) (count int64)
func InitRecord(typ types.Type) (record proto.Message)

// Reader instance
type Reader
    func Open(file string) (*Reader, error)
    func (r *Reader) Close() error
    func (r *Reader) Next(msg proto.Message) error
    func (r *Reader) ReadHeader() *types.Header
```

6.5.2 Reading Netcap Data

Reading *Netcap* audit records from disk can be done via the *netcap.Open* function:

```go
var (
    r, err = netcap.Open("test")
    packet = new(netcap.TCP)
)
if err != nil {
    panic(err)
}
defer r.Close()

for {
    err := r.Next(packet)
    if err == io.EOF {
        println("EOF")
        break
    } else if err != nil {
        panic(err)
    }

    fmt.Println(packet)
}
```

6.5.3 Dataset Labeling

To create a labeled dataset, first the *Netcap* dump files must be generated from a PCAP input source. If capture should happen live from an interface, wireshark or tcpdump should be used to obtain the PCAP file, as this file is needed for being scanned with suricata to obtain the alerts.

Collect traffic from interface:

```
$ tcpdump -w traffic.pcap -i eth0
```

After obtaining traffic, collect netcap audit records from PCAP dump file:

```
$ netcap -r traffic.pcap
```

Afterwards, the input file can be scanned and results mapped onto all generated data. For this purpose, the initially used PCAP dump file must be supplied, along with the **-label** command-line flag. Note for this step, the suricata IDS must be installed and configured with a ruleset on the analysis machine. Emerging Threats offers a extensive public ruleset, that is also recommended by the suricata authors. Using the suricata-update script for installing the ruleset, offers a convient and fast way of installing the rules and keeping them up to date. [sur18] The version of the ruleset used for the experiments contained a total of 23947 rules, of which 19017 were enabled. After setting everything up, the following *Netcap* command must be executed to generate the labeled data:

Create labels:

```
$ netcap -r traffic.pcap -label
```

The emitted files are encoded as CSV and will have the *_labeled.csv* file extension. A new column will be appended to the end, named 'result'. This column contains a value for each row, either 'normal' or the classification of the observed malicious behavior.

6.6 Deep Learning with Tensorflow

TensorFlow is an open source software framework accelerated numerical computation, and was released by Google in 2015. It features a flexible architecture, that allows deployment of computation procedures across many different platforms. These include CPUs, GPUs, TPUs, and deployment options range from desktop computers, to server clusters and mobile devices. *Tensorflow* comes with powerful capabilities for machine learning and deep neural networks, although its flexible numerical computation core is utilized across various other scientific domains. *Tensorflow* is currently available in version 1.12. On the development machine the latest supported version (1.12.0) for macOS is installed. It is important to note that there is no GPU support on macOS, so the CPU will be used for computation. Alternatively, Tensorflow could be executed within a docker container running linux. [ten18]

6.6.1 Introduction

Use of Tensorflow for classification of the KDD dataset has been demonstrated in the Washington University Course T81-558: Applications of Deep Neural Networks, by Prof. Jeff Heaton. [jef18] His approach will be adopted to work with labeled data produced by *Netcap*, which can be generated from any PCAP dump file or from live traffic.

Deep Learning

Deep Learning is a very popular type of machine learning, that is based upon the original neural networks developed in the 1980's. Todays implementations don't differ much from the original neural networks. The term deep neural networks simply refers to neural networks with multiple layers. While creation and calculation of deep neural networks has been possible for the last 20 years, effective means of training them have been a long time problem. Deep learning provides efficient means to train deep neural networks and solve this problem. [jef18]

Choosing a Dataset

Anomaly-based intrusion detection approaches suffer from the lack of reliable datasets for testing and verification. Evaluations of the eleven datasets that exist since 1998, conducted by the Canadian Institute of Cybersecurity, have shown that most datasets are out of date and unreliable for evaluation purposes. While some of these datasets fail to provide sufficient diversity and volume of network traffic, some do not contain latest attack patterns or anonymize packet payload data, which limits research capabilities. Additionally, some are also lacking feature set and metadata information. [cic18] An interesting approach towards the creation of realistic datasets was presented in 2015, when researchers from the TU Darmstadt released their framework for injecting attacks into existing datasets. [CVM+15] Although their tool was not used in this research, it is an promising option for future experiments.

CIC IDS 2017 Dataset

The *CIC-IDS-2017* dataset has been chosen as the most up to date and most extensively documented dataset, and will be used for the following experiments. The dataset consists of CSV flow records generated with CICFlowMeter, and the PCAP files used for creation. Data was captured for a total of five days and no anonymization has been performed on the dataset. The total size of all included PCAP dumpfiles is 48,9 GB. Attacks have been carried out on the working days Tuesday, Wednesday, Thursday and Friday, in both morning and afternoon. The implemented attacks include DoS & DDoS (Wednesday), Heartbleed (Wednesday), Web Attacks (Thursday), FTP Brute Force & SSH Brute Force (Tuesday), Infiltration (Thursday), Botnet traffic and Port Scans (Friday). Monday is the normal day and includes only legitimate traffic. Detailed information on the dataset and the testbed architecture can be found in the corresponding research paper [SHLG18] and on the CIC website [cic18].

6.6.2 Feature Collection

To get an impression about the data we are dealing with, the capinfos tool from *wireshark* can be used:

```
$ capinfos Thursday-WorkingHours.pcap
File name:           Thursday-WorkingHours.pcap
File type:           Wireshark/... - pcapng
File encapsulation:  Ethernet
File timestamp precision:  microseconds (6)
Packet size limit:   file hdr: (not set)
Number of packets:   9322 k
File size:           8302 MB
Data size:           7992 MB
Capture duration:    29145.871747 seconds
First packet time:   2017-07-06 13:58:58.492265
Last packet time:    2017-07-06 22:04:44.364012
Data byte rate:      274 kBps
Data bit rate:       2193 kbps
Average packet size: 857.39 bytes
Average packet rate: 319 packets/s
SHA256:              38f8b1bb276849bf1721f7c4de22bebfa7f59a74e52286d4c0a37edbb118fe01
RIPEMD160:           e34ba9cf32ee15838585564d881ff1415fb8beb6
SHA1:                21450a49b206d60107d16e0918f26e4b74afadad
Strict time order:   False
Capture oper-sys:    Linux 4.8.0-22-generic
Capture application: mergecap
Number of interfaces in file: 1
Interface #0 info:
                Encapsulation = Ethernet (1 - ether)
                Capture length = 262144
                Time precision = microseconds (6)
                Time ticks per second = 1000000
                Number of stat entries = 0
                Number of packets = 9322025
```

It can be seen that the dump actually has the PCAPNG format, despite its *.pcap* extension. Baselayer is Ethernet, capture was performed from one interface, average packet rate is 319 packets/s, average packet size is 857.39 bytes and the total size of the dump file is 8302 MB.

Now feature collection with *Netcap* can begin. The default configuration will be used, output will be written in a separate folder, specified with the **-out** flag.

```
$ netcap -r Thursday-WorkingHours.pcap -out Thursday-WorkingHours -workers 1000
                            / |
    _______    ______   _10 |_     _______   ______    ______
    /     / \  /    / \ / 01/  |   /     / | /    / \  /    / \
    0010100 /|/011010 /|101010/   /0101010/  001010  |/100110  |
    01 |  00 |00    00 |  10 | __ 00 |       /    10 |00 |  01 |
    10 |  01 |01001010/   00 |/  |01 \_____ /0101000 |00 |__10/|
    10 |  00 |00/    / |  10  00/ 00/    / |00    00 |00/   00/
    00/   10/  0101000/    0010/   0010010/  0010100/ 1010100/
                                                        00 |
    Network Protocol Analysis Framework               00 |
    created by Philipp Mieden, 2018                   00/
    v0.3.2

    +---------------+--------+
    |   Setting     | Value  |
    +---------------+--------+
    | Workers       | 1000   |
    | MemBuffer     | true   |
    | Compression   | true   |
    | PacketBuffer  | 100    |
    +---------------+--------+

    opening Thursday-WorkingHours.pcap | size: 8.3 GB
    counting packets... done. 9322025 packets found in 8.154473734s
    spawned 1000 workers
    initialized 29 layer encoders | buffer size: 4096
    initialized 7 custom encoders | buffer size: 4096
    done.

    Processed 9322025 packets (7992634369 bytes) in 11m50.325536666s (errors: 1456226, type:6)
    Final flush: 870 closed

    TCP stats:

    +-----------------------+-------------+
    |      Description      |   Value     |
    +-----------------------+-------------+
    | IPdefrag              | 92          |
    | missed bytes          | 4012192822  |
    | total packets         | 3387720     |
    | rejected FSM          | 105624      |
    | rejected Options      | 1347376     |
    | reassembled bytes     | 3292334514  |
    | total TCP bytes       | 7390883110  |
    | conn rejected FSM     | 58312       |
    | reassembled chunks    | 527546      |
    | out-of-order packets  | 2051020     |
    | out-of-order bytes    | 2763558926  |
    | biggest-chunk packets | 1487        |
    | biggest-chunk bytes   | 1287117     |
    | overlap packets       | 135828      |
    | overlap bytes         | 2934887     |
    +-----------------------+-------------+

    Errors: 1456244

    +----------------------+----------+
    |        Error         |  Count   |
    +----------------------+----------+
    | FSM                  | 105624   |
    | OptionChecker        | 1347376  |
    | HTTP-response-body  | 3028      |
    | HTTP-request-body   | 33        |
    | HTTP-response       | 176       |
    | HTTP-request        | 7         |
    +--------------------+----------+

    flushed 0 http events. requests 54122 responses 37007

    -> total bytes of data written to disk: 518 MB
    -> 0.09766% of packets (9104) written to unknown.pcap
    -> 0.00009% of packets (8) written to errors.pcap


    +-------------------------------+-------------+-------------+
    |           Protocol            | NumPackets  |   Share     |
    +-------------------------------+-------------+-------------+
    | Ethernet                      | 9322025     | 100.00000%  |
    | SNAP                          | 7344        | 0.07878%    |
    | CiscoDiscovery                | 7344        | 0.07878%    |
    | IPv4                          | 9240723     | 99.12785%   |
    | TCP                           | 8538148     | 91.59113%   |
    | DNS                           | 644796      | 6.91691%    |
    | NTP                           | 14160       | 0.15190%    |
    | ICMPv6NeighborAdvertisement   | 8           | 0.00009%    |
    | Geneve                        | 2           | 0.00002%    |
    | UDP                           | 726608      | 7.79453%    |
    | ICMPv6NeighborSolicitation    | 182         | 0.00195%    |
    | ICMPv4                        | 1429        | 0.01533%    |
    | VXLAN                         | 2           | 0.00002%    |
    | Payload                       | 4517774     | 48.46344%   |
    | ICMPv6                        | 2342        | 0.02512%    |
    | MLDv2MulticastListenerReport  | 1660        | 0.01781%    |
    | DecodeFailure                 | 6           | 0.00006%    |
    | ARP                           | 45829       | 0.49162%    |
    | IPv6                          | 28030       | 0.30069%    |
    | DHCPv6                        | 20927       | 0.22449%    |
    | LinkLayerDiscoveryInfo        | 99          | 0.00106%    |
    | LLC                           | 7344        | 0.07878%    |
    | ICMPv6RouterSolicitation      | 492         | 0.00528%    |
    | LinkLayerDiscovery            | 99          | 0.00106%    |
    | IGMP                          | 130         | 0.00139%    |
    | IPv6HopByHop                  | 1660        | 0.01781%    |
    | Fragment                      | 96          | 0.00103%    |
    +-------------------------------+-------------+-------------+
    Active Custom Encoders:
    + TLS
    + LinkFlow
    + NetworkFlow
    + TransportFlow
    + HTTP
    + Flow
    + Connection


    [ERROR] BFD packet length does not match COUNT: 2
    [ERROR] invalid first SIP line: 'staticccmbgcom,staticccmbgcomedgekeynet.5e12525d' COUNT: 2
    [ERROR] Unable to decode EthernetType 26996 COUNT: 2
    [ERROR] GTP packet too small: 76 bytes COUNT: 2

    done in 11m50.423536122s
```

The following files were generated:

```
$ du -h Thursday-WorkingHours/*
256K    Thursday-WorkingHours/ARP.ncap.gz
 16M    Thursday-WorkingHours/Connection.ncap.gz
128K    Thursday-WorkingHours/DHCPv6.ncap.gz
 13M    Thursday-WorkingHours/DNS.ncap.gz
107M    Thursday-WorkingHours/Ethernet.ncap.gz
 30M    Thursday-WorkingHours/Flow.ncap.gz
2.1M    Thursday-WorkingHours/HTTP.ncap.gz
 12K    Thursday-WorkingHours/ICMPv4.ncap.gz
 12K    Thursday-WorkingHours/ICMPv6.ncap.gz
4.0K    Thursday-WorkingHours/ICMPv6NeighborAdvertisement.ncap.gz
4.0K    Thursday-WorkingHours/ICMPv6NeighborSolicitation.ncap.gz
4.0K    Thursday-WorkingHours/ICMPv6RouterSolicitation.ncap.gz
4.0K    Thursday-WorkingHours/IGMP.ncap.gz
160M    Thursday-WorkingHours/IPv4.ncap.gz
192K    Thursday-WorkingHours/IPv6.ncap.gz
8.0K    Thursday-WorkingHours/IPv6HopByHop.ncap.gz
 48K    Thursday-WorkingHours/LLC.ncap.gz
8.0K    Thursday-WorkingHours/LinkFlow.ncap.gz
4.0K    Thursday-WorkingHours/LinkLayerDiscovery.ncap.gz
4.0K    Thursday-WorkingHours/LinkLayerDiscoveryInfo.ncap.gz
384K    Thursday-WorkingHours/NTP.ncap.gz
2.1M    Thursday-WorkingHours/NetworkFlow.ncap.gz
 48K    Thursday-WorkingHours/SNAP.ncap.gz
171M    Thursday-WorkingHours/TCP.ncap.gz
4.1M    Thursday-WorkingHours/TLS.ncap.gz
9.1M    Thursday-WorkingHours/TransportFlow.ncap.gz
 10M    Thursday-WorkingHours/UDP.ncap.gz
4.0K    Thursday-WorkingHours/errors.pcap
4.1M    Thursday-WorkingHours/unknown.pcap
 20K    Thursday-WorkingHours/errors.log
```

This step has to be repeated for all files in the dataset that should be analyzed. Note, that the Monday file will not be further evaluated since on Monday no attacks were performed. Table 6.1 displays a summary of the processing with *Netcap*:

File	Num Packets	Size	Processing Time	Extracted Data
Tuesday-WorkingHours.pcap	11551954	11 GB	11m11.918614271s	440 MB
Wednesday-WorkingHours.pcap	13788878	13 GB	12m58.13498869s	536 MB
Thursday-WorkingHours.pcap	9322025	8.3 GB	8m39.274062535s	369 MB
Friday-WorkingHours.pcap	9997874	8.8 GB	9m0.653867719s	410 MB

Table 6.1: Statistics for processing input pcaps

6.6.3 Labeling

To label the generated data, the *netlabel* tool must be used. After the *suricata* scan on the input file completed, a summary table of alerts will be shown. Then, labeling of each generated audit record will be performed asnychronously for each audit record type. Once complete and if labels were applied, the record will be added to the summary.

Example for Thursday-WorkingHours.pcap

```
$ netlabel -r Thursday-WorkingHours.pcap -out Thursday-WorkingHours
scanning Thursday-WorkingHours.pcap with suricata...
done. reading logs from Thursday-WorkingHours/fast.log
got 29913 labels

+-----------------------------------------+--------+
|             Classification              | Count  |
+-----------------------------------------+--------+
| Not Suspicious Traffic                  | 90     |
| Potentially Bad Traffic                 | 5      |
| A Network Trojan was detected           | 23737  |
| Successful Administrator Privilege Gain | 5      |
| Attempted Information Leak              | 2      |
| Misc Attack                             | 22     |
| Potential Corporate Privacy Violation   | 6      |
| Misc activity                           | 4      |
| Web Application Attack                  | 96     |
| Generic Protocol Command Decode         | 5946   |
+-----------------------------------------+--------+

+ Connection_labeled.csv                  100% labels: 1870
+ DNS_labeled.csv                         100% labels: 5
+ Ethernet_labeled.csv                    100% labels: 27147
+ Flow_labeled.csv                        100% labels: 4121
+ HTTP_labeled.csv                        100% labels: 5093
+ IPv4_labeled.csv                        100% labels: 27147
+ NTP_labeled.csv                         100% labels: 22
+ TCP_labeled.csv                         100% labels: 27120
+ TransportFlow_labeled.csv               100% labels: 2
+ UDP_labeled.csv                         100% labels: 27

done in 7m7.841170332s
```

The following files were generated:

```
$ du -h Thursday-WorkingHours/*.csv
57M     Thursday-WorkingHours/Connection_labeled.csv
128M    Thursday-WorkingHours/DNS_labeled.csv
704M    Thursday-WorkingHours/Ethernet_labeled.csv
128M    Thursday-WorkingHours/Flow_labeled.csv
15M     Thursday-WorkingHours/HTTP_labeled.csv
880M    Thursday-WorkingHours/IPv4_labeled.csv
2.0M    Thursday-WorkingHours/NTP_labeled.csv
1.2G    Thursday-WorkingHours/TCP_labeled.csv
8.0M    Thursday-WorkingHours/TransportFlow_labeled.csv
40M     Thursday-WorkingHours/UDP_labeled.csv
```

6.6.4 Labeling Results

The following section shows the alerts generated by suricata for each file of the dataset, as well as a summary of the total number of labels per file and the time it took for *Netcap* to map them onto the derived data.

Tuesday-WorkingHours.pcap

Classification	Count
Generic Protocol Command Decode	6449
Attempted Information Leak	3
Not Suspicious Traffic	80
Potential Corporate Privacy Violation	12
Potentially Bad Traffic	25
Misc Attack	18

Table 6.2: Suricata alerts for file Tuesday-WorkingHours.pcap

Wednesday-WorkingHours.pcap

Classification	Count
Not Suspicious Traffic	50
Potential Corporate Privacy Violation	5
Misc Attack	25
Attempted Information Leak	2
Web Application Attack	8
Potentially Bad Traffic	15
Generic Protocol Command Decode	8095

Table 6.3: Suricata alerts for file Wednesday-WorkingHours.pcap

Thursday-WorkingHours.pcap

Classification	Count
Generic Protocol Command Decode	5946
Potential Corporate Privacy Violation	6
Attempted Information Leak	2
A Network Trojan was detected	23761
Web Application Attack	96
Misc Attack	23
Not Suspicious Traffic	90
Potentially Bad Traffic	5
Successful Administrator Privilege Gain	5
Misc activity	4

Table 6.4: Suricata alerts for file Thursday-WorkingHours.pcap

Friday-WorkingHours.pcap

Classification	Count
Not Suspicious Traffic	115
Attempted Information Leak	3
Potentially Bad Traffic	30
Misc activity	4
Generic Protocol Command Decode	6016
Potential Corporate Privacy Violation	21

Table 6.5: Suricata alerts for file Friday-WorkingHours.pcap

Statistics

Table 6.6 displays the number of alerts for each file and the total processing time for mapping the alerts to the corresponding audit records with *Netcap*:

File	Labels	Processing Time
Tuesday-WorkingHours.pcap	6656	29m 33s
Wednesday-WorkingHours.pcap	7793	52m 56s
Thursday-WorkingHours.pcap	29912	2h 13m 21s
Friday-WorkingHours.pcap	5626	40m 53s

Table 6.6: Statistics for label generation with *Netcap* v0.3.2

6.7 TensorFlow Deep Neural Network

6.7.1 Overview

The implementation of the Deep Neural Network uses the *Keras* toolkit, the created model has the type *sequential*, which is a linear stack of layers. Five layers are being added with *kernel_initializer* type *normal* and *activation* type *relu*. The input shape is specified via the *input_dim* parameter and is set dynamically. The model is compiled with the loss function *categorical_crossentropy* and the *adam* optimizer.

6.7.2 Preparations

Jeff Heatons implementation needed to be modified in order to work with data produced by *Netcap*. First, explicitely setting the expected columns was removed, as CSV data generated by *Netcap* contains the correct fields as first line.

Because *Netcap* data can contain boolean values, a function was added to encode them to numeric values:

```python
# this creates a boolean series
# casting to int converts True and False to 1 and 0 respectively
def encode_bool(df, name):
    print(colored("encode_bool␣" + name, "yellow"))
    df[name] = df[name].astype(int)
```

To allow encoding also if there are missing values (NaN), the *encode_text_index* and *encode_numeric_zscore* functions were slightly modified. Prior to encoding, all missing values will be replaced with the zero value for the expected data type, that is an empty string literal for *encode_text_index*, and 0 for *encode_numeric_zscore*.

```python
# Encode text values to indexes(i.e. [1],[2],[3] for red,green,blue).
def encode_text_index(df, name):
    # replace missing values (NaN) with an empty string
    df[name].fillna('',inplace=True)
    print(colored("encode_text_index␣" + name, "yellow"))
    le = preprocessing.LabelEncoder()
    df[name] = le.fit_transform(df[name])
    return le.classes_

# Encode a numeric column as zscores
def encode_numeric_zscore(df, name, mean=None, sd=None):
    # replace missing values (NaN) with a 0
    df[name].fillna(0,inplace=True)
    print(colored("encode_numeric_zscore␣" + name, "yellow"))
    if mean is None:
        mean = df[name].mean()

    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd
```

The function *encode_string* decides at runtime which method to pick for encoding alphanumeric values:

```
def encode_string(df, name):
"""
Encode string decides which method for encoding strings will be called.
"""
if arguments.string_index:
    encode_text_index(df, col)
if arguments.string_dummy:
    encode_text_dummy(df, col)
```

To enhance readability of the generated output, a primitive for coloring strings was imported:

```
    from termcolor import colored
```

To display the size of the input file, two utility functions were added:

```
def convert_bytes(num):
    for x in ['bytes', 'KB', 'MB', 'GB', 'TB']:
        if num < 1024.0:
            return "%3.1f␣%s" % (num, x)
        num /= 1024.0


def file_size(file_path):
    if os.path.isfile(file_path):
        file_info = os.stat(file_path)
        return convert_bytes(file_info.st_size)
```

To make the program configurable from the command-line, flags have been added. Printing the help with the **-h** flag yields the following output:

```
usage: netcap-tf-dnn.py [-h] -read READ [-drop DROP] [-sample [SAMPLE]]
                        [-dropna] [-string_dummy] [-string_index]
                        [-test_size TEST_SIZE] [-loss LOSS]
                        [-optimizer OPTIMIZER]

NETCAP compatible implementation of Network Anomaly Detection with a Deep
Neural Network and TensorFlow

optional arguments:
-h, --help            show this help message and exit
-read READ            Labeled input CSV file to read from (required)
-drop DROP            optionally drop specified columns, supply multiple
                        with comma
-sample [SAMPLE]      optionally sample only a fraction of records
-dropna               drop rows with missing values
-string_dummy         encode strings as dummy variables
-string_index         encode strings as indices (default)
-test_size TEST_SIZE  specify size of the test data in percent (default:
                        0.25)
-loss LOSS            set function (default: categorical_crossentropy)
-optimizer OPTIMIZER  set optimizer (default: adam)
```

In the following, example usage of the command-line flags is demonstrated:

```
$ python netcap-tf-dnn.py -r Connection_labeled.csv -sample 0.5 -drop SrcIP,DstIP
$ python netcap-tf-dnn.py -r Connection_labeled.csv -string_dummy -dropna
$ python netcap-tf-dnn.py -r Connection_labeled.csv -dropna -drop SrcMAC
```

Installation can be performed using the provided *install.sh* script from the project root.

6.7.3 Encoding of Features

Features need to be encoded to numeric values for each column of the dataset. Numeric values are normalized with the standard score, also called zscore, which is the signed number of standard deviations by which the value of the current data point is above the mean value of all observations. Observed values below the mean have negative standard scores, while values above the mean have positive standard scores. Alphanumeric values (strings) will be encoded to dummy variables. For example the strings red, green and blue will be encoded as [1,0,0],[0,1,0] and [0,0,1]. Encoding takes a while, depending on the size of the dataset. To enable working with different types of datasets, an encoder dictionary was added, to lookup column names and provide the appropriate encoder function. In the future, this could be refactored to infer the datatype and the appropriate encoder for the datatype, based on the first line of values, instead of having to define the explicitely for each column name. Features that are guaranteed to be unique for each record, namingly the UID of LayerFlows and the SessionID of TLS handshakes, are always dropped from the dataset.

The following shows an excerpt of the implemented encoders dictionary in python. Field names are mapped to the desired encoding function.

```python
# encoder dictionary
encoders = {
    # Flow / Connection
    'TimestampFirst'   : encode_numeric_zscore,
    'LinkProto'        : encode_string,
    'NetworkProto'     : encode_string,
    'TransportProto'   : encode_string,
    'ApplicationProto' : encode_string,
    'SrcMAC'           : encode_string,
    'DstMAC'           : encode_string,
    'SrcIP'            : encode_string,
    'SrcPort'          : encode_numeric_zscore,
    'DstIP'            : encode_string,
    'DstPort'          : encode_numeric_zscore,
    'Size'             : encode_numeric_zscore,
    'AppPayloadSize'   : encode_numeric_zscore,
    'NumPackets'       : encode_numeric_zscore,
    'UID'              : encode_string,
    'Duration'         : encode_numeric_zscore,
    'TimestampLast'    : encode_numeric_zscore,

    # UDP specific fields
    'Length'           : encode_numeric_zscore,
    'Checksum'         : encode_numeric_zscore,
    'PayloadEntropy'   : encode_numeric_zscore,
    'PayloadSize'      : encode_numeric_zscore,
    'Timestamp'        : encode_numeric_zscore,
    ...
}
```

6.7.4 Training & Validation

For training the deep neural network, a test / train split is created. A share of 25% is used as test dataset for verification. To avoid overfitting, training the model will be stopped, once the validation accuracy starts to decrease. After verification the program prints out the validation score. The validation score represents the accuracy of the models classifications, and can be interpreted as a percentage value. For example, a validation score of 1.0 means all labels have been classified correctly. The following python code creates the test split, performs training and runs validation:

```python
print("breaking␣into␣predictors␣and␣prediction...")

# Break into X (predictors) & y (prediction)
x, y = to_xy(df,'result')

print("creating␣train/test␣split")

# Create a test/train split.  25% test
# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25, random_state=42)

print("creating␣neural␣network...")

# Create neural network
model = Sequential()
model.add(Dense(10, input_dim=x.shape[1], kernel_initializer='normal', activation='relu'))
model.add(Dense(50, input_dim=x.shape[1], kernel_initializer='normal', activation='relu'))
model.add(Dense(10, input_dim=x.shape[1], kernel_initializer='normal', activation='relu'))
model.add(Dense(1, kernel_initializer='normal'))
model.add(Dense(y.shape[1],activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
monitor = EarlyStopping(
    monitor='val_loss',
    min_delta=1e-3,
    patience=5,
    verbose=1,
    mode='auto'
)

print("fitting␣model...")
model.fit(
    x_train,
    y_train,
    validation_data=(x_test,y_test),
    callbacks=[monitor],
    verbose=2,
    epochs=1000
)

print("measuring␣accuracy...")
pred = model.predict(x_test)
pred = np.argmax(pred,axis=1)
y_eval = np.argmax(y_test,axis=1)
score = metrics.accuracy_score(y_eval, pred)

print("Validation␣score:␣{}".format(colored(score, 'yellow')))
print("Exec␣Time:␣{}".format(colored(display_time(time.time() - start_time), 'yellow')))
print("done.")
```

6.7.5 Scripts

Since these steps have to be repeated for each file in the dataset, scripts have been created for executing the experiments. The following shows the bash script **run_exp1.sh** for executing experiment 1:

```bash
#!/bin/bash

NUM=1

echo "[INFO]␣running␣experiment␣$NUM"
echo "[INFO]␣parsing␣data"
netcap -r Tuesday-WorkingHours.pcap -out Tuesday-WorkingHours-$NUM
netcap -r Wednesday-WorkingHours.pcap -out Wednesday-WorkingHours-$NUM
netcap -r Thursday-WorkingHours.pcap -out Thursday-WorkingHours-$NUM
netcap -r Friday-WorkingHours.pcap -out Friday-WorkingHours-$NUM

echo "[INFO]␣labeling␣data"
netlabel -r Tuesday-WorkingHours.pcap -out Tuesday-WorkingHours-$NUM
netlabel -r Wednesday-WorkingHours.pcap -out Wednesday-WorkingHours-$NUM
netlabel -r Thursday-WorkingHours.pcap -out Thursday-WorkingHours-$NUM
netlabel -r Friday-WorkingHours.pcap -out Friday-WorkingHours-$NUM

echo "[INFO]␣evaluating"
eval.sh Tuesday-WorkingHours-$NUM -string_dummy
eval.sh Wednesday-WorkingHours-$NUM -string_dummy
eval.sh Thursday-WorkingHours-$NUM -string_dummy
eval.sh Friday-WorkingHours-$NUM -string_dummy

echo "[INFO]␣stats"
stats.sh Tuesday-WorkingHours-$NUM
stats.sh Wednesday-WorkingHours-$NUM
stats.sh Thursday-WorkingHours-$NUM
stats.sh Friday-WorkingHours-$NUM

echo "[INFO]␣done."
```

The *eval.sh* script receives a path to the directory containing the labeled CSV files, and calls the python program to read, encode, train and validate each file and save the log.

```bash
#!/bin/bash

if [[ $1 == "" ]]; then
    echo "[ERROR]␣need␣a␣path␣as␣argument"
    exit 1
fi

for f in $(ls $1/*_labeled.csv); do
    echo "[INFO]␣processing␣$f"
    # "${@:2}" passes down all arguments after the directory name
    netcap-tf-dnn.py -read $f "${@:2}" | tee "${f%_labeled.csv}_RESULT.txt"
done
```

The *stats.sh* script parses the generated result files and displays a table summary. The experiments are started from the datasets PCAPs directory, using the UNIX time program to track execution time. Output is piped into tee, which displays it on the terminal and writes it into the named logfile on disk for later analysis.

```
$ time run_exp1.sh | tee run_exp1.log
```

6.7.6 Classification Results Experiment 1

For experiment number one, all numeric values will be zscored. Strings will be encoded as dummy variables, which creates a new column for each unique entry and deletes the original column from the dataset afterwards. Booleans are encoded to numeric values, 0 for false, 1 for true. The first run of experiment 1 was stopped after 12 hours, after hanging for more than 7 hours at the dummy variable encoding of the Answers columns from Tuesday-WorkingHours-1/DNS_labeled.csv. In order to reduce the amount of data that has to be processed, the sample size was dropped from 1.0 to 0.5. This means only 50% of the available data will be used. However, the next run also did not complete in a reasonable amount of time, and was stopped after 8 hours, hanging at the same place like before. After starting a third run, while sampling only 20% of the available data, the experiment finished after over 29 hours. Experiment one was conducted with an early version of *Netcap*, and was not repeated. It is shown here for completeness and because following experiments were adapted after observations from this one. Entries marked with ”-” contained only one label type after sampling, thus no prediction was possible and the *Keras* library aborted.

Tuesday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	284166	51 MB	2119	50s	0.9919065381096488
DNS_labeled.csv	700447	144 MB	33	3h 27m	0.9999428946692174
Ethernet_labeled.csv	11551954	880 MB	3556	6m 49s	0.9996693201846267
Flow_labeled.csv	647294	112 MB	4852	4m 2s	0.9904835470415573
HTTP_labeled.csv	45852	14 MB	5609	21s	0.9637554585152839
IPv4_labeled.csv	11469736	1.0 GB	3555	-	-
NTP_labeled.csv	15507	2.1 MB	18	2s	0.9987113402061856
TCP_labeled.csv	10710230	1.5 GB	3504	-	-
TransportFlow_labeled.csv	91861	5.8 MB	11	8s	0.9999059354717336
UDP_labeled.csv	787015	44 MB	51	27s	0.9999491753703845

Table 6.7: Classification results experiment 1 Tuesday-WorkingHours.pcap

Wednesday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
ARP_labeled.csv	45243	3.5 MB	1	-	-
Connection_labeled.csv	333653	60 MB	8408	59s	0.9757837319426962
DNS_labeled.csv	714164	145 MB	22	2h 52m	0.9999719958553865
Ethernet_labeled.csv	13788878	1.0 G	4420	7m 40s	0.9997070102865497
Flow_labeled.csv	1029945	192 MB	29583	7m 35s	0.9707755640995767
HTTP_labeled.csv	212324	56 MB	171933	36m 24s	0.9877439426793627
IPv4_labeled.csv	13705555	1.3 GB	4419	-	-
NTP_labeled.csv	12624	2.0 MB	25	1s	0.9968354430379747
TCP_labeled.csv	12943316	1.8 GB	4372	-	-
TLS_labeled.csv	12943316	23 MB	1	-	-
TransportFlow_labeled.csv	95768	6.1 MB	8	8s	1.0
UDP_labeled.csv	787951	44 MB	47	27s	0.9999746180009138

Table 6.8: Classification results experiment 1 Wednesday-WorkingHours.pcap

Thursday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	317531	57 MB	1870	52s	0.993449644139321
DNS_labeled.csv	644796	128 MB	5	2h 5m	1.0
Ethernet_labeled.csv	9322025	704 MB	27169	5m 37s	0.9989186916168564
Flow_labeled.csv	680753	128 MB	4121	4m 7s	0.9947999294905694
HTTP_labeled.csv	54221	15 MB	5090	23s	0.9678729689807977
IPv4_labeled.csv	9240723	880 MB	27169	-	-
NTP_labeled.csv	14160	2.0 MB	22	1s	0.9971751412429378
TCP_labeled.csv	8538148	1.2 GB	27142	-	-
TransportFlow_labeled.csv	116637	8.0 MB	3	-	-
UDP_labeled.csv	726608	40 MB	27	27s	0.9991742588973 61

Table 6.9: Classification results experiment 1 Thursday-WorkingHours.pcap

Friday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	445148	80 MB	1971	1m 21s	0.9940246203612184
DNS_labeled.csv	668172	128 MB	36	2h 27m	0.9999401358915262
Ethernet_labeled.csv	9997874	753 MB	3413	5m 39s	0.9996259206951874
Flow_labeled.csv	1091900	192 MB	4363	6m 50s	0.9963183441707116
HTTP_labeled.csv	140888	23 MB	97062	57s	0.985939497230507
IPv4_labeled.csv	9915680	928 MB	3413	-	-
TCP_labeled.csv	9191727	1.3 GB	3377	-	-
TransportFlow_labeled.csv	167351	12 MB	21	11s	1.0
UDP_labeled.csv	750104	42 MB	36	25s	0.9999466751986349

Table 6.10: Classification results experiment 1 Friday-WorkingHours.pcap

Observations

Using the dummy variable method for encoding strings delivered a good detection accuracy of around 99%, however it is impractical for operating on large datasets, due to the very high time for encoding of features. Because each unique string will lead to the creation of a new column, the dataset grows in width tremendously. For the evaluation of KDD-99 this strategy of encoding strings might have been possible, but regarding the size of modern datasets, another approach is needed. Dropping columns with a high amount of unique entries (such as DNS Answers, IP adresses) was not considered, because this first series of experiments aims to create a baseline of detection accuracy with the complete data that *Netcap* delivers. Entries with ”-” have failed when entering the training phase, because after sampling and encoding, the subset did only contain normal labels and none of the malicious types.

6.7.7 Classification Results Experiment 2

For experiment number two, encoding will be slightly modified in order to reduce the time needed for the encoding stage. All numeric values will be zscored like before. Booleans are encoded to numeric values, 0 for false, 1 for true. Instead of choosing dummy variables, strings will be encoded as indices this time. This keeps the original column in the dataset and creates a new one containing a unique integer for each unique string. Originally this was used to encode only the result column, the following experiment evaluates the use of this string encoding method for the whole dataset. Experiment 2 had a total processing time of over 8 hours.

Tuesday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	284166	51M	1807	49s	0.9936375665099518
DNS_labeled.csv	700447	144M	38	1m 57s	0.9999600255836265
Ethernet_labeled.csv	11551954	880M	3549	33m 41s	0.9996984060534857
Flow_labeled.csv	647294	112M	3340	1m 51s	0.9946855843385406
HTTP_labeled.csv	45800	14M	4214	38s	0.9275109170305676
NTP_labeled.csv	15507	3.0M	11	4s	0.9989682744389993
NetworkFlow_labeled.csv	29094	4.0M	1291	15s	0.9573824580698378
TCP_labeled.csv	10710230	1.5G	3450	29m 59s	0.9996728362186739
TransportFlow_labeled.csv	212613	20M	1721	44s	0.99153403318659
UDP_labeled.csv	787015	43M	44	2m 2s	0.9999440926232758

Table 6.11: Classification results experiment 2 Tuesday-WorkingHours.pcap

Wednesday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	333653	60M	2223	54s	0.9937060925024577
DNS_labeled.csv	714164	144M	49	2m 10s	0.9999551923647789
Ethernet_labeled.csv	13788878	1.0G	4147	38m 16s	0.9996965670888426
Flow_labeled.csv	1029945	190M	3493	3m 21s	0.9965396311270084
HTTP_labeled.csv	70243	20M	27675	45s	0.9412903593189453
NTP_labeled.csv	12624	2.0M	19	3s	0.9987325728770595
NetworkFlow_labeled.csv	29495	4.0M	1361	15s	0.9555193924599946
TCP_labeled.csv	12943316	1.8G	3649	36m 31s	0.9997150652892968
TransportFlow_labeled.csv	212502	20M	2159	48s	0.9896096073485675
UDP_labeled.csv	787951	43M	33	2m 3s	0.9999796941945702

Table 6.12: Classification results experiment 2 Wednesday-WorkingHours.pcap

Thursday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	317531	57M	1784	54s	0.9940163511079199
DNS_labeled.csv	644796	122M	9	1m 44s	0.9999813894627138
Ethernet_labeled.csv	9322025	701M	27145	27m 60s	0.9970607254129681
Flow_labeled.csv	680753	128M	3189	3m 5s	0.9955461281281399
HTTP_labeled.csv	54137	15M	3265	44s	0.949907646841522
NTP_labeled.csv	14160	2.0M	20	3s	0.9994350282485875
NetworkFlow_labeled.csv	27921	4.0M	1206	10s	0.9610371007019052
TCP_labeled.csv	8538148	1.2G	27066	27m 43s	0.9989238884123348
TransportFlow_labeled.csv	256349	23M	1691	46s	0.99344651104731
UDP_labeled.csv	726608	40M	23	2m 7s	0.9999779798736045

Table 6.13: Classification results experiment 2 Thursday-WorkingHours.pcap

Friday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	445148	80M	1561	1m 27s	0.99656743375237
DNS_labeled.csv	668172	128M	38	2m 6s	0.9999521081398203
Ethernet_labeled.csv	9997874	752M	3163	36m 17s	0.9996703299780874
Flow_labeled.csv	1091900	192M	2841	3m 20s	0.9973074457367891
HTTP_labeled.csv	113138	20M	4032	46s	0.9664840021212657
NetworkFlow_labeled.csv	25837	3.0M	1137	11s	0.9600619195046439
TCP_labeled.csv	9191727	1.3G	3088	33m 47s	0.9996596940205367
TransportFlow_labeled.csv	311675	29M	1395	1m 19s	0.9952771467806312
UDP_labeled.csv	750104	41M	35	2m 7s	0.9999360088734363

Table 6.14: Classification results experiment 2 Friday-WorkingHours.pcap

Observations

Processing time for encoding is now much better. Detection accuracy on HTTP traffic declined, most notably on the Tuesday-WorkingHours dump. Others slightly increased, for example the Tuesday-WorkingHours/Connection_labeled.csv detection accuracy increased from 99,19% to 99,21%. Accuracy for detecting flows stayed almost the same.

6.7.8 Classification Results Experiment 3

For experiment number three, encoding will be the same as in experiment two. However, for labeling this experiment will use the attack descriptions generated by suricata, instead of the attack classes. Furthermore all lines that contain missing numeric values (NaNs) will dropped from the dataset. Alphanumeric values are not affected by this, because for example an HTTP request that has no referrer should also be taken into consideration. Experiment 3 had a processing time of over 9 hours. By checking the logs for the number of dropped entries, it can be seen that no data was discarded.

Labeling Results

The following new labels were generated and mapped onto the data:

Tuesday-WorkingHours.pcap

Classification	Count
SURICATA TLS invalid record/traffic	3078
ET POLICY Vulnerable Java Version 1.8.x Detected	1
ET POLICY curl User-Agent Outbound	1
SURICATA HTTP Request abnormal Content-Encoding header	6
SURICATA HTTP gzip decompression failed	17
ET TOR Known Tor Relay/Router (Not Exit) Node Traffic group 455	18
SURICATA TLS error message encountered	26
ET DNS Query to a *.pw domain - Likely Hostile	24
SURICATA TLS invalid record type	16
ET POLICY DNS Query For XXX Adult Site Top Level Domain	9
SURICATA HTTP unable to match response to request	233
SURICATA HTTP Host header invalid	1
SURICATA TLS invalid record version	3072
ET POLICY Python-urllib/ Suspicious User Agent	2
ET POLICY GNU/Linux APT User-Agent Outbound likely related to package management	80
ET POLICY PE EXE or DLL Windows file download HTTP	3

Table 6.16: Suricata alerts for file Tuesday-WorkingHours.pcap

Wednesday-WorkingHours.pcap

Classification	Count
ET CURRENT_EVENTS Possible OpenSSL HeartBleed Large HeartBeat Response (Client Init Vuln Server)	1
ET CURRENT_EVENTS Possible OpenSSL HeartBleed Large HeartBeat Response (Server Init Vuln Client)	1
SURICATA TLS invalid record version	2896
ET POLICY PE EXE or DLL Windows file download HTTP	3
SURICATA HTTP unable to match response to request	2313
SURICATA TLS invalid certificate	2
ET DNS Query for .su TLD (Soviet Union) Often Malware Related	8
SURICATA TLS invalid record/traffic	2928
SURICATA HTTP invalid response chunk len	6
SURICATA TLS error message encountered	34
ET POLICY Self Signed SSL Certificate (SomeOrganizationalUnit)	2
SURICATA HTTP Request abnormal Content-Encoding header	2
ET DNS Query to a *.pw domain - Likely Hostile	5
ET POLICY GNU/Linux APT User-Agent Outbound likely related to package management	48
ET POLICY Python-urllib/ Suspicious User Agent	2
ET POLICY OpenSSL Demo CA - Internet Widgits Pty (O)	2
SURICATA TLS invalid record type	34
SURICATA TLS invalid SSLv2 header	2
ET TOR Known Tor Relay/Router (Not Exit) Node Traffic group 166	25
SURICATA HTTP gzip decompression failed	33
ET WEB_CLIENT Possible HTTP 500 XSS Attempt (External Source)	8
SURICATA TLS invalid handshake message	2

Table 6.18: Suricata alerts for file Wednesday-WorkingHours.pcap

Thursday-WorkingHours.pcap

Classification	Count
SURICATA TLS invalid record/traffic	2802
ET TOR Known Tor Relay/Router (Not Exit) Node Traffic group 166	21
ET POLICY GNU/Linux APT User-Agent Outbound likely related to package management	90
ET POLICY Possible IP Check api.ipify.org	1
ET CURRENT_EVENTS Possible Phishing Redirect Dec 13 2016	2
ET SCAN Behavioral Unusual Port 135 traffic Potential Scan or Infection	4
SURICATA HTTP gzip decompression failed	22
SURICATA HTTP Request abnormal Content-Encoding header	2
ET POLICY Vulnerable Java Version 1.8.x Detected	2
SURICATA HTTP invalid response chunk len	3
ET POLICY Python-urllib/ Suspicious User Agent	2
SURICATA TLS error message encountered	32
ET DNS Query for .su TLD (Soviet Union) Often Malware Related	3
GPL EXPLOIT Microsoft cmd.exe banner	5
ET TROJAN Windows dir Microsoft Windows DOS prompt command exit OUTBOUND	3
ET SCAN Possible Nmap User-Agent Observed	96
ET POLICY SSLv3 outbound connection from client vulnerable to POODLE attack	1
SURICATA TLS invalid record version	2802
ET POLICY PE EXE or DLL Windows file download HTTP	4
SURICATA HTTP unable to match response to request	269
ET TOR Known Tor Relay/Router (Not Exit) Node Traffic group 455	1
SURICATA TLS invalid record type	14
ET TROJAN Windows Microsoft Windows DOS prompt command Error not recognized	23731

Table 6.20: Suricata alerts for file Thursday-WorkingHours.pcap

Friday-WorkingHours.pcap

Classification	Count
SURICATA TLS invalid record type	17
SURICATA TLS invalid handshake message	4
SURICATA HTTP invalid response chunk len	8
ET POLICY DNS Query For XXX Adult Site Top Level Domain	6
ET POLICY OpenSSL Demo CA - Internet Widgits Pty (O)	4
ET POLICY Possible IP Check api.ipify.org	9
ET POLICY GNU/Linux APT User-Agent Outbound likely related to package management	111
ET POLICY Python-urllib/ Suspicious User Agent	3
ET POLICY Vulnerable Java Version 1.8.x Detected	1
ET SCAN Behavioral Unusual Port 1433 traffic Potential Scan or Infection	2
ET DNS Query for .su TLD (Soviet Union) Often Malware Related	5
SURICATA TLS invalid record version	2790
ET POLICY PE EXE or DLL Windows file download HTTP	6
SURICATA HTTP gzip decompression failed	27
SURICATA TLS error message encountered	24
ET SCAN Behavioral Unusual Port 1434 traffic Potential Scan or Infection	2
SURICATA TLS invalid record/traffic	2793
SURICATA HTTP Request abnormal Content-Encoding header	5
ET DNS Query to a *.pw domain - Likely Hostile	24
SURICATA HTTP unable to match response to request	346

Table 6.22: Suricata alerts for file Friday-WorkingHours.pcap

Experiment 3 Results

Tuesday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	284166	51M	1803	1m 21s	0.9929900622167168
DNS_labeled.csv	700447	145M	38	2m 48s	0.999965736214537
Ethernet_labeled.csv	11551954	880M	3547	43m 36s	0.9996973672683657
Flow_labeled.csv	647294	111M	3334	1m 57s	0.9947412003163931
HTTP_labeled.csv	45795	14M	4206	38s	0.9336186566512359
NTP_labeled.csv	15507	2.1M	16	6s	0.9987103430487491
NetworkFlow_labeled.csv	29094	4.0M	1288	6s	0.9586197415452296
TCP_labeled.csv	10710230	1.5G	3444	29m 26s	0.9996728362186739
TransportFlow_labeled.csv	212613	20M	1715	34s	0.9918350453399556
UDP_labeled.csv	787015	43M	49	2m 1s	0.9999339276456896

Table 6.23: Classification results experiment 3 Tuesday-WorkingHours.pcap

Wednesday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	333653	60M	2095	1m 32s	0.99357422015489
DNS_labeled.csv	714164	145M	36	1m 56s	0.9999607933191816
Ethernet_labeled.csv	13788878	1.0G	3815	36m 28s	0.9997200642836837
Flow_labeled.csv	1029945	192M	3211	7m 15s	0.9970095577640813
HTTP_labeled.csv	70239	20M	27897	45s	0.9280182232346241
NTP_labeled.csv	12624	1.7M	21	4s	0.9987325728770595
NetworkFlow_labeled.csv	29495	4.0M	1338	12s	0.9560618388934092
TCP_labeled.csv	12943316	1.8G	3348	38m 20s	0.99974040655424
TransportFlow_labeled.csv	212502	20M	2042	40s	0.9907954673794376
UDP_labeled.csv	787951	43M	35	2m 18s	0.9999746177432128

Table 6.24: Classification results experiment 3 Wednesday-WorkingHours.pcap

Thursday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	317531	57M	1782	3m 0s	0.9942556970636031
DNS_labeled.csv	644796	129M	10	1m 60s	0.9999627789254276
Ethernet_labeled.csv	9322025	705M	27145	28m 10s	0.9970590090482457
Flow_labeled.csv	680753	128M	3187	4m 57s	0.9952347096463343
HTTP_labeled.csv	54131	15M	3262	35s	0.9462055715658021
NTP_labeled.csv	14160	1.9M	20	4s	0.9994350282485875
NetworkFlow_labeled.csv	27921	4.0M	1205	15s	0.9591749033089815
TCP_labeled.csv	8538148	1.2G	27067	26m 46s	0.9991942046448481
TransportFlow_labeled.csv	256349	23M	1691	1m 18s	0.9932904755960554
UDP_labeled.csv	726608	40M	23	2m 11s	0.9999779798736045

Table 6.25: Classification results experiment 3 Thursday-WorkingHours.pcap

Friday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	445148	79M	1523	1m 23s	0.9967651208137518
DNS_labeled.csv	668172	127M	38	2m 3s	0.9999640811048652
Ethernet_labeled.csv	9997874	750M	3109	30m 13s	0.9996779315926703
Flow_labeled.csv	1091900	181M	2776	7m 3s	0.99753090942394
HTTP_labeled.csv	113140	20M	4259	1m 41s	0.9663425844086971
NetworkFlow_labeled.csv	25837	2.9M	1107	9s	0.9580495356037152
TCP_labeled.csv	9191727	1.3G	3029	28m 19s	0.9996636105855178
TransportFlow_labeled.csv	311675	28M	1355	1m 26s	0.9956236604679218
UDP_labeled.csv	750104	41M	35	2m 13s	0.9999360088734363

Table 6.26: Classification results experiment 3 Friday-WorkingHours.pcap

Observations

Classification results are comparable to the ones from the previous run and in this case do not seem to be influenced by the higher amount of unique labels. This is quite interesting, because classifying specific attacks could generate more specific alerts, which would provide more value to the analyst.

6.7.9 Classification Results Experiment 4

In experiment 4, the description will be used for labeling again and strings are encoded as indices, same as in experiment 3. Missing numeric values will not be dropped, but instead all columns containing source or destination IP adresses will be dropped from the dataset, in order to observe the effect on classification accuracy. Experiment 4 had a processing time of 12 hours.

Tuesday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	284166	51M	1803	1m 31s	0.9936516426902395
DNS_labeled.csv	700447	134M	36	2m 9s	0.999965736214537
Ethernet_labeled.csv	11551954	870M	3550	34m 58s	0.9996987523151923
Flow_labeled.csv	647294	111M	3334	2m 24s	0.9947720980818667
HTTP_labeled.csv	45823	13M	4206	31s	0.9397695530726257
NTP_labeled.csv	15507	2.1M	17	6s	0.9987103430487491
NetworkFlow_labeled.csv	29094	4.0M	1288	17s	0.9558702227110256
TCP_labeled.csv	10710230	1.5G	3444	32m 53s	0.9996720892694014
TransportFlow_labeled.csv	212613	20M	1715	58s	0.9916657260036874
UDP_labeled.csv	787015	43M	50	2m 7s	0.9999288451568964

Table 6.27: Classification results experiment 4 Tuesday-WorkingHours.pcap

Wednesday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	333653	60M	2199	1m 3s	0.9933943942263889
DNS_labeled.csv	714164	135M	43	2m 15s	0.9999383895015711
Ethernet_labeled.csv	13788878	1.0G	4033	42m 38s	0.9996933761117655
Flow_labeled.csv	1029945	177M	3393	3m 19s	0.9966561418634727
HTTP_labeled.csv	70260	20M	28191	18s	0.9161969826359238
NTP_labeled.csv	12624	1.7M	20	3s	0.9987325728770595
NetworkFlow_labeled.csv	29495	4.0M	1361	21s	0.9534852183346895
TCP_labeled.csv	12943316	1.8G	3539	39m 17s	0.9997197008865425
TransportFlow_labeled.csv	212502	20M	2151	1m 6s	0.9899672476753378
UDP_labeled.csv	787951	43M	36	2m 10s	0.9999746177432128

Table 6.28: Classification results experiment 4 Wednesday-WorkingHours.pcap

Thursday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	317531	56M	1782	57s	0.9944950430192863
DNS_labeled.csv	644796	121M	8	1m 53s	0.9999689824378564
Ethernet_labeled.csv	9322025	702M	27138	26m 36s	0.9970654454159545
Flow_labeled.csv	680753	117M	3187	2m 3s	0.9957459060221283
HTTP_labeled.csv	54151	14M	3264	34s	0.9497710149209632
NTP_labeled.csv	14160	1.9M	20	6s	0.9994350282485875
NetworkFlow_labeled.csv	27921	3.2M	1205	12s	0.9561667382896433
TCP_labeled.csv	8538148	1.2G	27059	32m 57s	0.9992209083281293
TransportFlow_labeled.csv	256349	23M	1692	1m 12s	0.9937117713144427
UDP_labeled.csv	726608	40M	23	3m 14s	0.9999779798736045

Table 6.29: Classification results experiment 4 Thursday-WorkingHours.pcap

Friday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	445148	79M	1500	2m 6s	0.9963967040175402
DNS_labeled.csv	668172	127M	37	2m 56s	0.9999640811048652
Ethernet_labeled.csv	9997874	750M	3055	45m 39s	0.9996871335471654
Flow_labeled.csv	1091900	181M	2732	4m 37s	0.9974759593369357
HTTP_labeled.csv	113141	20M	3838	46s	0.9645761153927738
NetworkFlow_labeled.csv	25837	2.9M	1091	17s	0.9586687306501548
TCP_labeled.csv	9191727	1.3G	2981	31m 20s	0.9996710085415931
TransportFlow_labeled.csv	311675	28M	1333	1m 54s	0.9954183190236015
UDP_labeled.csv	750104	41M	35	2m 21s	0.9999360088734363

Table 6.30: Classification results experiment 4 Friday-WorkingHours.pcap

Observations

Classification accuracy only slightly decreased for most audit record types, probably because identification of communicating clients is still possible by their hardware (MAC) addresses. Since the HTTP audit record type currently does not preserve the hardware addresses, it was more affected by this than other audit record types.

6.7.10 Classification Results Experiment 5

In experiment 5, the attack classes will be used again, encoding is the same as in experiment 2. The attack class "Generic Protocol Command Decode" (general decoding errors, i.e. invalid TLS handshakes) will be excluded, in order to observe the effect on accuracy. Experiment 5 had a processing time of 7 hours.

Tuesday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	284166	51M	67	53s	0.9998170096562596
DNS_labeled.csv	700447	133M	33	2m 0s	0.999965736214537
Ethernet_labeled.csv	11551954	868M	121	31m 1s	0.9999885733636797
Flow_labeled.csv	647294	111M	49	1m 47s	0.9999258453628633
HTTP_labeled.csv	45800	13M	1461	16s	0.9939737991266375
NTP_labeled.csv	15507	2.1M	13	5s	0.9989682744389993
NetworkFlow_labeled.csv	29094	3.3M	33	5s	0.9989001924663184
TCP_labeled.csv	10710230	1.5G	75	28m 47s	0.9999925305072757
TransportFlow_labeled.csv	212613	19M	59	33s	0.9996613613274636
UDP_labeled.csv	787015	43M	46	2m 1s	0.9999390101344826

Table 6.31: Classification results experiment 5 Tuesday-WorkingHours.pcap

Wednesday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	333653	60M	54	55s	0.9999040928381326
DNS_labeled.csv	714164	133M	14	1m 55s	0.9999831971367921
Ethernet_labeled.csv	13788878	1.0G	94	36m 43s	0.9999924576905449
Flow_labeled.csv	1029945	177M	33	2m 59s	0.9999572793966297
HTTP_labeled.csv	70243	19M	1444	12s	0.9789305848186322
NTP_labeled.csv	12624	1.7M	22	4s	0.9984157160963245
NetworkFlow_labeled.csv	29495	3.3M	35	5s	0.9987794955248169
TCP_labeled.csv	12943316	1.8G	58	36m 19s	0.9999412824348888
TransportFlow_labeled.csv	212502	19M	39	33s	0.9998117682490683
UDP_labeled.csv	787951	43M	36	2m 3s	0.9999746177432128

Table 6.32: Classification results experiment 5 Wednesday-WorkingHours.pcap

Thursday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	317531	57M	161	53s	0.999357545066324
DNS_labeled.csv	644796	129M	4	1m 39s	0.9999937964875713
Ethernet_labeled.csv	9322025	704M	23951	24m 22s	0.9974108638163284
Flow_labeled.csv	680753	128M	145	1m 57s	0.9998002221060116
HTTP_labeled.csv	54137	15M	1075	15s	0.9915035094200222
NTP_labeled.csv	14160	1.9M	20	4s	0.9994350282485875
NetworkFlow_labeled.csv	27921	4.0M	54	5s	0.9988540323735855
TCP_labeled.csv	8538148	1.2G	23911	23m 31s	0.9996013186934685
TransportFlow_labeled.csv	256349	23M	140	42s	0.9992198227437273
UDP_labeled.csv	726608	40M	23	1m 52s	0.9999779798736045

Table 6.33: Classification results experiment 5 Thursday-WorkingHours.pcap

Friday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	445148	80M	83	1m 14s	0.9998112987141355
DNS_labeled.csv	668172	129M	35	1m 47s	0.9999640811048652
Ethernet_labeled.csv	9997874	752M	156	27m 1s	0.999983996600878
Flow_labeled.csv	1091900	192M	68	3m 3s	0.9999303965564612
HTTP_labeled.csv	113138	20M	981	45s	0.9965352660420718
NetworkFlow_labeled.csv	25837	3.0M	42	4s	0.9979876160990712
TCP_labeled.csv	9191727	1.3G	118	24m 12s	0.9999921668700379
TransportFlow_labeled.csv	311675	29M	76	47s	0.9997304893543295
UDP_labeled.csv	750104	42M	35	1m 56s	0.9999360088734363

Table 6.34: Classification results experiment 5 Friday-WorkingHours.pcap

Observations

Apparently this improved the classification accuracy for the HTTP audit record type. Accuracy on other audit record types looks also promising. We can assume this alert type created a lot of noise in previous experiments.

6.7.11 Classification Results Experiment 6

In experiment 6, the attack description will be used again, all alerts for the attack class "Generic Protocol Command Decode" will be exlcuded and encoding is the same as in experiment 2. If there are multiple classifications for the same audit record, the label will collect all of them, i.e. if a TCP record matches "Suspicious Traffic" and "Web Application Attack", the final label will be: "Suspicious Traffic | Web Application Attack". Classifications will not be duplicated, i.e. in case there two alerts for "Suspicious Traffic" it will only be added once to the label. Experiment 6 had a processing time of 9,5 hours.

Tuesday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	284166	51M	68	50s	0.9996340193125194
DNS_labeled.csv	700447	144M	33	1m 57s	0.999965736214537
Ethernet_labeled.csv	11551954	880M	121	29m 39s	0.999989265887093
Flow_labeled.csv	647294	112M	49	1m 52s	0.9999320249159581
HTTP_labeled.csv	45790	13M	1461	17s	0.9951956673654787
NTP_labeled.csv	15507	3.0M	13	4s	0.9987103430487491
NetworkFlow_labeled.csv	29094	4.0M	33	5s	0.9984877646411878
TCP_labeled.csv	10710230	1.5G	75	27m 26s	0.9999925305072757
TransportFlow_labeled.csv	212613	20M	59	46s	0.9996613613274636
UDP_labeled.csv	787015	43M	46	1m 57s	0.9999440926232758

Table 6.35: Classification results experiment 6 Tuesday-WorkingHours.pcap

Wednesday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	333653	60M	51	60s	0.9998441508619657
DNS_labeled.csv	714164	144M	14	1m 50s	0.9999831971367921
Ethernet_labeled.csv	13788878	1.0G	87	34m 37s	0.9999930378681953
Flow_labeled.csv	1029945	192M	33	2m 47s	0.9999456283229833
HTTP_labeled.csv	70241	19M	1444	13s	0.9785889186265019
NTP_labeled.csv	12624	2.0M	15	3s	0.9990494296577946
NetworkFlow_labeled.csv	29495	4.0M	34	5s	0.9987794955248169
TCP_labeled.csv	12943316	1.8G	58	33m 9s	0.9999941282434888
TransportFlow_labeled.csv	212502	20M	39	34s	0.9998305914241614
UDP_labeled.csv	787951	43M	29	1m 54s	0.9999847706459277

Table 6.36: Classification results experiment 6 Wednesday-WorkingHours.pcap

Thursday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	317531	57M	161	53s	0.9994205308441354
DNS_labeled.csv	644796	128M	3	1m 39s	0.9999937964875713
Ethernet_labeled.csv	9322025	704M	23970	24m 35s	0.9974061438133419
Flow_labeled.csv	680753	128M	145	1m 55s	0.9998060979264229
HTTP_labeled.csv	54134	15M	1075	16s	0.9916506576030737
NTP_labeled.csv	14160	2.0M	20	5s	0.9994350282485875
NetworkFlow_labeled.csv	27921	4.0M	53	5s	0.9969918349806618
TCP_labeled.csv	8538148	1.2G	23931	22m 4s	0.9996003817221252
TransportFlow_labeled.csv	256349	23M	140	38s	0.9994070652852328
UDP_labeled.csv	726608	40M	23	1m 52s	0.9999779798736045

Table 6.37: Classification results experiment 6 Thursday-WorkingHours.pcap

Friday-WorkingHours.pcap

File	Num Records	Size	Labels	Exec Time	Validation Score
Connection_labeled.csv	445148	80M	84	1m 17s	0.9998292702651702
DNS_labeled.csv	668172	128M	35	1m 45s	0.9999640811048652
Ethernet_labeled.csv	9997874	752M	156	40m 5s	0.9999835965159
Flow_labeled.csv	1091900	192M	68	5m 2s	0.9999267332173276
HTTP_labeled.csv	113140	20M	981	51s	0.99717164574863
NetworkFlow_labeled.csv	25837	3.0M	42	8s	0.9989164086687307
TCP_labeled.csv	9191727	1.3G	118	39m 45s	0.9999917316961512
TransportFlow_labeled.csv	311675	29M	77	1m 10s	0.9997818247154096
UDP_labeled.csv	750104	41M	35	2m 42s	0.9999360088734363

Table 6.38: Classification results experiment 6 Friday-WorkingHours.pcap

Observations

Regarding the high accurarcy in the obtained results for all audit record types, this way of dealing with multiple alerts for the same audit records looks very promising and should be subject to further experiments. With this approach, all information that was collected when labeling the data is preserved and therefore is made available to the analyst.

6.7.12 Summary

Results show that accurate classification of malicious or unwanted behavior is possible, with a feature set consisting mainly of collected and only a handful of generated features. Furthermore classification achieved better results when applied to specific protocols, compared to classification of summary structures such as flows and connections. The time needed to encode the data to a numeric feature vector, heavily depends on the strategy used for encoding alphanumeric values (strings). Encoding strings as indices performed better in the encoding phase, but required more time for training the deep neural network, compared to encoding them as dummy variables. Using attack descriptions instead of attack classes as labels delivered comparable results and should be subject to further experiments. Due to the fact that there might be several alerts for the same audit records, a strategy for handling this must be chosen. Collecting all classifications and merging them into a final label has shown great results, and should be further investigated in combination with using specific attack descriptions instead of attack classes.

7 Conclusion

The implemented solution is referred to as *secure*, because parsing the network traffic is implemented in a memory-safe language. It is *scalable* because *Netcap* can increase throughput with more processing cores and can be used in a distributed monitoring setup. Tensorflow can take advantage of multi-core architectures as well as GPUs, in order to accelerate the computations for the Deep Neural Network. Results from the series of experiments show that classification of labeled audit records is possible with a high accuracy, while only using a reduced feature subset compared to previous research in the area. *Netcap* is meant to be a useful tool in the process of feature collection, selection and generation for research on anomaly-based intrusion detection strategies. By providing the generated data in separate files, *Netcap* enables other applications that consume its output to implement a concurrent design as well. Choosing a platform neutral format for serialized structured data, greatly increases accessibility of the generated audit records for experiments across all major programming languages. The landscape of interesting libraries and techniques is constantly evolving, and so are the possibilities and options for experiments. Implementing the framework in Go helps in reducing syntactic complexity, increases performance compared to implementations in scripting languages, and provides memory safety. The fact that memory safety is a big problem for todays intrusion detection frameworks has been recognized by the authors of *suricata*, which have begun to port parts of their protocol decoding logic to *Rust*, and the authors of *Bro / Zeek* that use a parser generation framework to accomplish safe protocol parsing. Although, these countermeasures are a good step forward, they only reduce the attack surface partially, since the remaining parts of the frameworks are still written in a language that is affected by this category of vulnerabilities. A key takeaway from the series of experiments is the absence of many computationally expensive extracted features, while still achieving very high classification results. Questioning the need for those features comes with the possibility to improve the performance of network monitoring systems. The fewer data is needed for classification of malicious behavior, the more efficient implementations for feature collection and extraction can become. By publishing the developed tooling, the experiments are made reproducible for other researchers, which will hopefully lead to more academic publications and novel research in the area. However, the problem is not and will never be solved simply by deploying an intrusion detection system - there is always the need for an analyst that monitors the alerts and events and takes appropriate action. It is important to keep in mind, that network monitoring systems can be evaded as well and may even provide an increased attack surface to the system or network they were supposed to protect. To counter this development, frequent updates to defensive strategies and tools are necessary.

8 Future Perspectives

Future development on *Netcap* will target increasing the unit test coverage, as well as adding more benchmarks for performance critical operations. Additionally, the output of *Netcap* will be compared to those of other tools, to ensure no data is missed or misinterpreted. *Netcap* will be extended with further functionality in the future, for example by adding support for extracted features that have proven value in other academic publications. Furthermore, *Netcap* should be used for experiments on other datasets, to explore its capabilities in delivering the necessary intelligence for accurate predictions on network data. Encoding of the feature vectors could also be implemented as part of the *Netcap* framework, as this would tremendously speed up processing compared to conducting encoding in a scripting language. An interface for adding additional application layer encoders that require stream reassembly is also planned. The project will be evaluated for monitoring industrial control systems communication, as recent efforts have been made by contributors to *gopacket* to implement support for Modbus, Ethernet/IP (ENIP), and the Common Industrial Protocol (CIP). A DPI module could help identify app layer protocols, before discarding the packet payloads. Another interesting candidate for future integration, is the recently open sourced fingerprinting strategy for SSH handshakes (HASSH) by salesforce. An in-depth performance analysis of *Netcap* will be carried out, as many parts of the framework can be further optimized. Robustness tests need to be performed and error handling improved to be more graceful and prevent the monitor from crashing in edge cases. Several exporters for the most common SIEM and data analytics solutions will be added, in order to visualize and organize data produced by *Netcap* for human analysts. Protocol coverage will be increased, as there are still protocols offered by *gopacket* that are not yet integrated into *Netcap*, i.e: *IPSec*, *MLDv2MulticastListenerReport*, *Geneve*, *VXLAN*, *USB* and more. To satisfy the need for large scale monitoring, tooling needs to be created to manage a multi instance architecture and deliver specific traffic to selected instances. The Canadian Institute for Cybersecurity meanwhile has published their newest dataset from 2018 (CSE-CIC-IDS2018). This dataset has an overall size of almost 500GB and will be evaluated with *Netcap* in the future as well.

The need for accurate and efficient intrusion detection systems is even greater today than it was in the early 1980s, when initial research started and the first frameworks were born. Next generation intrusion detection systems should be completely created in a secure programming language, and offer extensibility and scalability to meet todays demands for monitoring of complex large scale networks. Although network security monitoring is very effective, it is not the only security measure that can be put into place. A recent trend to secure networks is also the use of honeypots, which are intentionally vulnerable and serve as a trap for intruders. Placing those inside a cooperate network can help to identify an intruder in his early stages of lateral movement. Using honeypots as canaries is not a replacement for an intrusion detection system, but may serve well as an addition. The publication of a POSIX compliant kernel written in Golang [CKM18] as well as the first go network driver *ixy-go* [Voi18], are promising developments around the topic of memory safe system design. *Netcap* is not only restricted to network protocols - the *gopacket* library does also offer support for decoding USB traffic for example. Adding this functionality to *Netcap* can be done with little effort. However, maybe anomaly detection alone is not the solution to the problem: There should also be a critical discussion of the anomaly detection paradigm, as conducted by Gates et al. [GT07].

Glossary

Machine Learning The science of training computers to learn and behave like humans. By supplying data and information in the form of observations, the goal is to improve the learning process over time in autonomous fashion.

Artificial Intelligence Intelligence demonstrated by machines through implementing specific algorithms, mimicing natural intelligence displayed by humans and animals.

Data Mining The procedure of analyzing large volume data sets to recognize patterns and establish relationships between them, in order to solve problems or extract information from the input data set.

Turing Complete A computational system that is able to compute every Turing-computable function is referred to as Turing-complete (or Turing-powerful). Alternatively, a turing complete system is one that is able to simulate a universal Turing machine.

Deep Neural Network A set of algorithms designed to recognize patterns, inspired by the human brain. A deep neural network is basically an artificial neural network (ANN) with multiple layers between the input and output layers.

Acronyms

HTTP Hypertext Transfer Protocol

FTP File Transfer Protocol

SFTP Secure File Transfer Protocol

SMTP Simple Mail Transfer Service

DNS Domain Name System

TCP Transmission Control Protocol

UDP User Datagram Protocol

IP Internet Protocol

MTU Maximum Transmission Unit

TTL Time To Live

TLS Transport Layer Security

SSL Secure Socket Layer

NAT Network Address Translation

VPN Virtual Private Network

POP3 Post Office Protocol

IMAP Internet Message Access Protocol

ICMP Internet Control Message Protocol

SIP Session Initiation Protocol

RPC Remote Procedure Call

JSON Javascript Object Notation

FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
TPM	Trusted Platform Module
MAC	Media Access Control
PAT	Port Address Translation
VHDL	Very High Speed Integrated Circuit Hardware Description Language
IDEA	International Data Encryption Algorithm
WMR	Weight by Maximum Relevance
ASIC	Application-Specific Integrated Circuit
RAM	Random Access Memory
IoT	Internet of Things
DMA	Direct Memory Access
DGA	Domain Generation Algorithms
MD5	Message Digest Algorithm 5
CSV	Comma-separated values
AV	Anti Virus
HTTPS	Hypertext Transfer Protocol Secure
BitTorrent	BitTorrent File Transfer Protocol
DMZ	Demilitarized Zone
VR	Virtual Reality
LoC	Lines of Code
PCAP	Packet Capture
PCAPNG	Packet Capture Next Generation
OS	Operating System

DPI	Deep Packet Inspection
IPS	Intrusion Prevention System
IDS	Intrusion Detection System
API	Application Programming Interface
NIC	Network Interface Card
URL	Uniform Resource Locator
REST	Representational State Transfer
SOC	Security Operation Centers
IoC	Indicators Of Compromise
LRZ	Leibniz Rechenzentrum
MWN	Münchener Wissenschaftsnetzwek
DHCP	Dynamic Host Configuration Protocol
GRE	Generic Routing Encapsulation
SSH	Secure Shell
FIDS	Flow-based Intrusion Detection System
OSI	Open Systems Interconnection
LDA	Linear Discriminant Analysis
SIEM	Security Information and Event Management
IDES	Intrusion Detection Expert System
CIDF	Common Intrusion Detection Framework
FUD	Fully Undetectable
CPU	Central Processing Unit
IPFIX	Internet Protocol Flow Information Export
IETF	Internet Engineering Task Force

NSM Network Security Monitoring

NSA National Security Agency

IRC Internet Relay Chat

QUIC Quick UDP Internet Connections

BPF Berkeley Packet Filter

eBPF Berkeley Packet Filter

RFC Request For Comments

SVD Singular Value Decomposition

PCA Principal Component Analysis

NGO Non-Governmental Organisation

OISF Open Information Security Foundation

IEEE Institute of Electrical and Electronics Engineers

ISO International Institute for Standardization

ML Machine Learning

AI Artificial Intelligence

ANN Artificial Neural Network

HMM Hidden Markov Models

GA Genetic Algorithms

GP Genetic Programming

SVM Support Vector Machines

DoS Denial Of Service

DDoS Distributed Denial Of Service

MitM Man in the Middle

List of Figures

List of Tables

Bibliography

[aci18] ACI: *Securing IoT Devices.* http://www.theamericanconsumer.org/wp-content/uploads/2018/09/FINAL-Wi-Fi-Router-Vulnerabilities.pdf, 10 2018. – accessed 1 November 2018

[apl18] LGTM: *Kernel RCE caused by buffer overflow in Apple's ICMP packet-handling code (CVE-2018-4407).* https://lgtm.com/blog/apple_xnu_icmp_error_CVE-2018-4407, 2018. – accessed 18 October 2018

[Bac00] Bace, Rebecca G.: *Intrusion Detection.* 1. Indianapolis : Macmillan Technical Publishing USA, 2000. – ISBN 1–57870–185–6

[BCV01] Biermann, E. ; Cloete, E. ; Venter, L.M.: A comparison of Intrusion Detection systems. In: *Computers and Security, 20* (2001), S. 676–683

[Bej13] Bejtlich, Richard: *The Practice Of Network Security Monitoring - Understanding Incident Detection and Response.* 6. San Francisco : No Starch Press, 2013. – ISBN 978–1–59327–509–9

[BK14] Bhattacharyya, Dhruba K. ; Kalita, Jugal K.: *Network Anomaly Detection: A Machine Learning Perspective.* 1. Boca Raton, Lodon, New York : CRC Press, 2014. – ISBN 978–1–4665–8208–8

[bro18a] Bro Tracker: *Parser Fuzzing.* https://bro-tracker.atlassian.net/browse/BIT-524, 2018. – accessed 9 December 2018

[bro18b] Bro Network Security Monitor: *release notes.* https://www.bro.org/sphinx/install/release-notes.html, 2018. – accessed 9 December 2018

[cap17] Cap'n Proto: *Compiler Optimization elides bounds check.* https://nvd.nist.gov/vuln/detail/CVE-2017-7892, 4 2017. – accessed 2 November 2018

[cic18] University of new Brunswick: *Intrusion Detection Evaluation Dataset (CICIDS2017).* https://www.unb.ca/cic/datasets/ids-2017.html, 2018. – accessed 5 November 2018

[cis18] Heise Security: *Zero-Day-Lücke in Cisco Adaptive Security Appliance und Firepower Threat Defense.* https://www.heise.de/security/meldung/Zero-Day-Luecke-in-Cisco-Adaptive-Security-Appliance-und-Firepower-Threat-Def html, 2018. – accessed 19 October 2018

[CK74] Cerf, Vinton G. ; Khan, Robert E.: A Protocol for Packet Network Intercommunication. In: *IEEE TRANSACTIONS ON COMMUNICATIONS* 22 (1974), S. 637–648

[CKM18] Cutler, Cody ; Kaashoek, M. F. ; Morris, Robert T.: The benefits and costs of writing a POSIX kernel in a high-level language. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA : USENIX Association, 2018. – ISBN 978–1–931971–47–8, 89–105

[Col17] Collins, Michael: *Network Security Through Data Analysis - From Data to Action*. 2. Sebastopol : O'Reilly, 2017. – ISBN 978–1–491–96284–8

[CVM+15] Cordero, C. G. ; Vasilomanolakis, E. ; Milanov, N. ; Koch, C. ; Hausheer, D. ; Mühlhäuser, M.: ID2T: A DIY dataset creation toolkit for Intrusion Detection Systems. In: *2015 IEEE Conference on Communications and Network Security (CNS)*, 2015, S. 739–740

[Den87] Denning, D. E.: An Intrusion-Detection Model. In: *IEEE Transactions on Software Engineering* SE-13 (1987), Feb, Nr. 2, S. 222–232. http://dx.doi.org/10.1109/TSE.1987.232894. – DOI 10.1109/TSE.1987.232894. – ISSN 0098–5589

[DH12] Davidoff, Sherri ; Ham, Jonathan: *Network Forensics: Tracking Hackers through Cyberspace*. 1. New York : Prentice Hall, 2012. – ISBN 978–0–13–256471–7

[fb218] Reuters: *Facebook says big breach exposed 50 million accounts to full takeover*. https://www.reuters.com/article/us-facebook-cyber/facebook-unearths-security-breach-affecting-50-million-users-idUSKCN1M82BK, 2018. – accessed 18 October 2018

[goc17] Ravelin: *So just how fast are channels anyway?* https://syslog.ravelin.com/so-just-how-fast-are-channels-anyway-4c156a407e45, 2017. – accessed 7 November 2018

[gop18] Google: *Gopacket Repository*. https://github.com/google/gopacket, 2018. – accessed 21 October 2018

[GT07] Gates, Carrie ; Taylor, Carol: Challenging the Anomaly Detection Paradigm: A Provocative Discussion. In: *Proceedings of the 2006 Workshop on New Security Paradigms*. New York, NY, USA : ACM, 2007 (NSPW '06). – ISBN 978–1–59593–923–4, 21–29

[hac18] Arstechnica: *How Hacking Team got hacked*. https://arstechnica.com/information-technology/2016/04/how-hacking-team-got-hacked-phineas-phisher/, 2018. – accessed 18 October 2018

[Hos09] Hossain, Moazzam: *Intrusion Detection with Artificial Neural Networks*. 1. Saarbrücken : Lambert Academic Publishing, 2009. – ISBN 978–3–639–21038–5

[ipf18] IETF: *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information*. https://tools.ietf.org/html/rfc7011, 2018. – accessed 29 November 2018

[IZ14] Iglesias, Felix ; Zseby, Tanja: Analysis of network traffic features for anomaly detection. In: *Mach Learn (2015), Springer* (2014), S. 59–84

[jav11] accessed 18 November 2018

[jav17] Synopsys: *Exploiting the Java Deserialization Vulnerability.* `https://www.synopsys.com/content/dam/synopsys/sig-assets/whitepapers/exploiting-the-java-deserialization-vulnerability.pdf`, 2017. – accessed 2 November 2018

[jef18] Washington University in St.Louis: *T81-558:Applications of Deep Neural Networks.* `https://sites.wustl.edu/jeffheaton/t81-558/`, 2018. – accessed 5 November 2018

[KKD+15] Kim, Young-Hwan ; Konow, Roberto ; Dujovne, Diego ; Turletti, Thierry ; Dabbous, Walid ; Navarro, Gonzalo: PcapWT: An efficient packet extraction tool for large volume network traces. In: *Computer Networks* 79 (2015), 91 - 102. `http://dx.doi.org/https://doi.org/10.1016/j.comnet.2014.12.007`. – DOI https://doi.org/10.1016/j.comnet.2014.12.007. – ISSN 1389–1286

[Luc10] Lucas, Michael W.: *Network Flow Analysis.* 2. San Francisco : No Starch Press, 2010. – ISBN 978–1–59327–203–6

[McN16] McNab, Chris: *Network Security Assessment.* 3. Sebastopol : O'Reilly, 2016. – ISBN 978–1–491–91095–5

[MSA05] Mukkamala, A. ; Sung, A. ; Abraham, A.: Cyber security challenges: Designing efficient intrusion detection systems and antivirus tools. (2005)

[mtr18] FireEye: *M Trends 2018.* `https://www.fireeye.com/content/dam/collateral/en/mtrends-2018.pdf`, 2018

[nvd18] NIST: *NVD Vulnerability Visualizations.* `https://nvd.nist.gov/general/visualizations/vulnerability-visualizations`, 2018

[Pat13] Pathak, Rupali: *Recent Advances in Intrusion Detection.* 1. Saarbrücken : Lambert Academic Publishing, 2013. – ISBN 978–3–659–29146–3

[Pat14] Pathan, Al-Sakib K.: *The State of the Art in Intrusion Prevention and Detection.* 1. Boca Raton, Lodon, New York : CRC Press, 2014. – ISBN 978–1–4822–0351–6

[Pax98] Paxson, Vern: Bro: A System for Detecting Network Intruders in Real-Time. In: *7th USENIX Security Symposium* (1998), January

[pca18] PCAP-NG: *PCAP-NG Specification.* `https://pcapng.github.io/pcapng/`, 2018. – accessed 20 October 2018

[pro18] Google: *Protocol Buffers Documentation.* `https://developers.google.com/protocol-buffers/`, 2018. – accessed 27 November 2018

[rat18] accessed 19 November 2018

[rob18] Golang: *Concurrency is not parallelism.* `https://blog.golang.org/concurrency-is-not-parallelism`, 2018. – accessed 5 November 2018

[SAH16] Sommer, Robin ; Amann, Johanna ; Hall, Seth: Spicy: a unified deep packet inspection framework for safely dissecting all your data. (2016), 12, S. 558–569. `http://dx.doi.org/10.1145/2991079.2991100`. – DOI 10.1145/2991079.2991100

[sFl03] sFlow: *Traffic Monitoring using sFlow.* `https://sflow.org/sFlowOverview.pdf`, 2003. – version 01.00, accessed 15 October 2018

[SHLG18] Sharafaldin, Iman ; Habibi Lashkari, Arash ; Ghorbani, Ali: Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization, 2018, S. 108–116

[Si16] So-in, Chakchai: *A Survey of Network Traffic Monitoring and Analysis Tools.* 2016

[SSK15] Stoffer, Vincent ; Sharma, Aashish ; Krous, Jay: 100G Intrusion Detection. 1 (2015), Aug, Nr. 1

[sur18] OISF Github: *suricata-update.* `https://github.com/OISF/suricata-update`, 2018. – accessed 5 November 2018

[SVDCP14] Sommer, Robin ; Vallentin, Matthias ; De Carli, Lorenzo ; Paxson, Vern: HILTI: An Abstract Execution Environment for Concurrent, Stateful Network Traffic Analysis. In: *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC* (2014), 11. `http://dx.doi.org/10.1145/2663716.2663735`. – DOI 10.1145/2663716.2663735

[ten18] Tensorflow: *Tensorflow Home Page.* `https://www.tensorflow.org`, 2018. – accessed 5 November 2018

[Ver18] Veracode: State of Software Security 2018 Report. (2018). – accessed 27 November 2018

[Voi18] Voit, Sebastian Peter J.: Writing Network Drivers in Go. (2018), 10

[wir18] Packet Foo: *Attacking Wireshark.* `https://blog.packet-foo.com/2018/09/attacking-wireshark/#more-2246`, 2018. – accessed 20 October 2018