University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Novel Artificial Intelligence Techniques and System Calls to Detect Heterogeneous Malware Affecting IoT Spectrum Sensors

Severin Kunz
Zürich, Switzerland
Student ID: 16-707-135

Supervisor: Dr. Alberto Huedras Celdrán, Jan Von Der Assen
Date of Submission: November 9, 2022

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

Die zunehmende Verbreitung von IoT-Geräten bietet Hackern neue Angriffsmöglichkeiten. Darüber hinaus erhöht die Konnektivität von IoT-Geräten das Schadenspotenzial von IoT-Systemen. Daher ist die Erkennung von Malware auf solchen Systemen entscheidend für die Schadensbegrenzung. Vor einigen Jahren hat das maschinelle Lernen in Kombination mit *behavioral fingerprinting*, bei dem Informationen vom Zustand des Geräts, wie z.B. die Speichernutzung oder Systemaufrufe genutzt werden, die dateibasierte Malware-Erkennung abgelöst. Diese Arbeit konzentriert sich auf die Erkennung von Malware auf der Grundlage von Systemaufrufen und enthält die folgenden Hauptbeiträge: Erstens, erweitert sie die Malware-Erkennung, indem sie die Klassifizierung spezifischer Angriffsphasen von Malware ermöglicht. Zweitens, wird das Potenzial von Deep-Learning-Modellen im Bereich der systemaufrufbasierten Erkennung von Angriffsphasen in IoT-Geräten evaluiert und mit einem neuronalen Netzwerk verglichen, das als Basismodell dient. Schließlich, wird in dieser Arbeit eine auf TF-IDF basierende, angepasste Preprocessing-Methode (*TF-DF*) für Systemaufrufe bewertet, die eine verbesserte statistische Zuordung der aussagekräftigsten Systemaufrufe anstrebt. Zu diesem Zweck wurde ein Datensatz erstellt, der aus Systemaufrufen besteht, die von einem Raspberry Pi stammen, das mit einem Hochfrequenznetzwerk verbunden ist. Aus den Systemaufrufen dieses Datensatzes wurden elf verschiedene Angriffsphasen, die von vier Malware-Typen (Backdoor, Botnet, Ransomware und Rootkit) stammen, und eine gutartige Phase abgeleitet. Die Klassifizierungsergebnisse des Neuronalen Netzwerkes haben die Ergebnisse der implementierten DL-Modelle in signifikanter Weise übertroffen. In Kombination mit der vorgeschlagenen Preprocessing-Methode *TF-DF* wurde bei dem Preprocessing von Systemaufrufsequenzen mit unterschiedlichen Längen ein F1-Score von 99,2% erzielt. In einem letzten Schritt, wurden die Modelle mit gleich langen Systemaufrufsequenzen evaluiert, wobei die TF-IDF Methode TF-DF übertraf und einen F1-Score von 92,9% erzielte.

Abstract

The spreading of IoT devices yields new attack vectors for hackers. In addition, the connectivity of IoT devices increases the potential damage to IoT systems. Therefore, detecting malware on such systems is crucial to limit the damage. Some years ago, Machine Learning combined with behavioral fingerprinting which takes information from the devices' state has superseded file-based malware detection. This thesis concentrates on system call based malware detection and entails the following main contributions: Firstly, it extends malware detection by enabling the classification of specific attack phases of malware. Secondly, it evaluates the potential of Deep Learning models in the area of system call based attack phase detection in IoT devices and compares it with a Neural Network serving as a baseline model. Finally, the thesis assesses a TF-IDF based adapted preprocessing technique (*TF-DF*) for system calls, that seeks an enhanced representation of the most expressive system calls. For these purposes, a dataset consisting of system calls coming from a Raspberry Pi connected to a radio frequency network has been created. From the system calls of this dataset, eleven different attack phases stemming from four malware types (backdoor, botnet, ransomware, and rootkit) and one benign phase have been deducted. The classification results of the Neural Network model have significantly outscored the results of the implemented DL models. In combination with the proposed preprocessing technique *TF-DF*, an F1-score of 99.2% has been achieved when applying it on system call sequences with differing lengths. In a final step, the models have been evaluated with receiving equal length system call sequences where TF-IDF outperformed TF-DF and yielded an F1-score of 92.9%.

Acknowledgments

I'm grateful to Dr. Alberto Huedras Celdrán, for giving me the opportunity to write my master's thesis with the Communication Systems Group led by Prof. Dr. Burkhard Stiller. His support throughout the thesis mostly provided within the bi-weekly meetings helped me a lot to get to this final version of the thesis.

Additionally, I want to thank Pedro Miguel Sánchez Sánchez for joining the meetings and sharing his knowledge of Machine and Deep Learning models. Further, I appreciate the support from Ramon Solo de Zaldivar who elaborately answered questions about the underlying dataset.

Contents

Chapter 1

Introduction

The number of IoT devices is rising constantly. According to the authors of [27], there have been 12.2 billion active devices at the end of 2021 which involves an increase of 8 percent in comparison to the previous year. There are a lot of interesting areas where IoT devices can be used, as integral parts of e.g. smart homes, smart gardening, or soon smart grids. Although there are many ways IoT devices could positively influence our lives, one has to be aware of the security risks that they bring along. On one hand, IoT devices create new attack vectors e.g. an attacker could shut down the electricity of a homeowner and demand ransom. On the other hand, IoT devices have fewer resources than standard PCs which makes them more vulnerable in comparison to standard PCs [38]. Moreover, due to the fact of limited resources anti-malware programs need to be lightweight and therefore might not deliver full protection.

As the installation of malware cannot always be prevented it is important to detect when a system is under attack. Further, not every infected system can be restored in the same way since the removal of malware differs in the type of malware that was executed. Therefore, knowing the installed malware type helps to be able to fastly repair a system. Furthermore, the connectivity of devices in the IoT world involves the danger of spreading malware to further devices. Therefore, the effectiveness of repair in the IoT realm is key.

It is indicated that the rise of artificial intelligence and natural language processing could improve the automatic detection of infected systems, especially when dealing with new malware programs [24]. Older static detection techniques were efficient in detecting already known malware by e.g. matching signatures (bytes or string specific to malware) with the current system state. But, finding malware that is not part of the signature database is often unsuccessful with such an approach since one of the strings respectively bytes have to map at least partly with the string or byte of the observed malware [1]. Combining AI models with dynamic and behavioral features yield a less strict detection. Therefore, this approach might increase the chance of correctly classifying systems that are infected by new malware programs. This could be achieved, by training a model with different traces of system calls that perform requests from the user space to the kernel space of a system [8]. In conclusion, it is not only important to detect malware in a non-whitelist approach but also to detect which malware infected a system to efficiently remove malware.

1.1 Motivation

In research of AI malware detection based on system calls the focus naturally lies on detecting if malware is installed on a system or not. The malware type detection based on system calls is still rarely researched especially in the realm of IoT devices. In addition, malware type classification on other devices was focusing on finding out which malware is installed on a system while not including a benign state of the system. This solution would therefore need a preliminary step to detect if the examined system is being under attack or not.

Furthermore, novel AI models such as Autoencoders and Transformers haven't been evaluated in the realm of system call based malware detection. Especially, Transformers seem to be promising as they are known to put different kinds of attention on pixels when it comes to classifying pictures [49]. Similarly, focusing on various aspects of system call traces could also help when classifying a system state. The further implemented LSTM model, an adaptation of the RNN model, also goes in a similar direction but focuses more on the importance of a given word within a specific sequence [20]. The third sequence model that has been implemented is the Convolutional Neural Network which especially puts attention to differences between neighboring pixels in a picture. Similarly, in this work, the CNN is applied to neighbouring arrays where an array consists of TF-(I)DF values of a system call sequence. Last but not least, a simple neural network has been implemented with a well-known (TF-IDF) and a newly established preprocessing (TF-DF) technique for natural language.

1.2 Description of Work

This work follows four different goals: First, it yields to do a summary of existing works using system calls to detect anomalies and malware on IoT devices and other systems (Windows, Linux & Android). Secondly, this thesis wants to extend the research of malware detection based on system calls by focusing on classifying malware-specific attack phases. By not only detecting the malware installed but also the attack phase of it the repairment of the system can be started with more detailed information and therefore could be done more efficiently. Thirdly, the mentioned research field is extended with new Machine Learning and Deep Learning models such as Autoencoders or Transformers and the TF-DF preprocessing technique that are evaluated and compared with existing malware classification results based on system calls. Whereas an Autoencoder is used to reduce the input to the most salient features the goal of using this model is to have a fast classification process, the sequence models (LSTM, Transformer & CNN) shall be used to learn the importance of a system call within a specific sequence. Further, a Neural Network is implemented to work as a baseline. The goal of the newly created TF-DF preprocessing technique is to map the distribution of system call frequencies more precisely than the TF-IDF method which is more optimal in the realm of natural language. Lastly, the detection of attack phases while detecting anomalies is evaluated in this work by adding a benign class to the classification process. Therefore, there would be no need

to have running an anomaly detection model and a malware or attack phase detection model at the same time.

1.3 Thesis Outline

In the first section background information on various topics of interest to the underlying thesis is discussed: First, several malware types are briefly presented. Then, the necessity of system calls and their possible collection is discussed. After that, preprocessing techniques for natural language processing (NLP) are discussed followed by the explanation of Machine Learning models implemented in this and related work. The first chapter is concluded with a short presentation of the ElectroSense platform which is the provenance of the data used to classify malware in this work.

The second chapter is dedicated to the related work focusing on anomaly and malware detection based on system calls of IoT devices but also android smartphones or standard PCs. Also, it is discussed how Deep Learning models have been implemented in the related work. Based on this, the third chapter consists of implementation details of models, and preprocessing techniques are presented right after a thorough exploration of the underlying data. Given the implementation details, the fourth chapter is evaluating the results of the different implemented ML models including some experiments with adversarial attacks. Next, results are discussed in the fifth chapter whereas limitations are presented and translated into future steps that could be taken. In the last chapter, the results are summarized.

Chapter 2

Background

The background is focused on six main topics, namely: system calls, malware types, preprocessing techniques, Machine and Deep Learning models, the dataset used in this thesis, and the ElectroSense platform where the latter was gathered. The former topic is chosen as system calls are the basis for the thesis' malware detection. This strategy is a subcategory of behavioral fingerprinting which can also be executed on other systems' data such as resource consumption (e.g. CPU or memory usage). In addition, file-based malware detection exists but it has been pointed out that it has some flaws. The observed metadata of files including IP addresses or URLs that the file will connect to provide information on the good nature of a program. As this is sometimes known by attackers they mostly mask those strings to make the anti-malware program ineffective. Therefore, this thesis is focused on system calls and the use of its behavior-based method of malware detection.

2.1 Malware

New malware programs are created each day. To better understand newly created malware it is categorized into a malware types category. By comparing potential attacks but also the ways of installation one can divide the malware programs into malware types. Further, the programs can be divided into malware families where a malware family not only shares its malware type but also other characteristics (e.g. package names). Therefore, malware families can be system-specific whereas malware types can not.

In this chapter, four widespread malware types that are also present in the dataset used in this thesis and additional malware types are introduced. Table 2.1 shows the different installation procedures and strategies that attackers choose when making use of the four main malware types. However, the first contact of an attacker with a potential victim is mostly independent of the malware type. There are two widespread ways to gain access to a victim's computer: Either through social engineering or by delivering malicious code onto the system by taking advantage of its vulnerability e.g. insufficient security in the current version of the installed operating system [16].

Malware Type	Malware Example	Infection	Further Strategies	Possible Attacks while Malware is Active
Backdoor [10]	TheTick	Download and installation of backdoor	Obfuscation of future access via remote access tools or through the use of Tor browser	Accessing system to e.g. steal data
Botnet [16]	Bashlite	- Download and Installation of bot binary - Joining botnet with authentication and connection to Botmaster server	Connection to DNS Server to hide IP	- Botmaster connects to bot to give commands - Performing HOLD, UDP or TCP floods
Ransomware [41]	RansomwarePoC	- Connection to control server to get encryption key - Encryption of system or files	- Search of valuable files - Hindering user from restoring and rebooting the system	Improving locking after first installation
Rootkit [69]	Bdvl	Replacing UNIX utilities or installing automated script	- Hiding of running processes by opening TCD/UDP ports, registry keys, and stored files - Protection from its deletion	- Network scanning and sniffing etc. - Running script - Obfuscation processes

Table 2.1: Overview of installation and attack procedures of different malware

2.1.1 Backdoor

A backdoor is a malware program that is installed on a victim's computer or network to provide unauthorized access to the attacker. For IoT devices, authorized access is probably installed by the seller of the devices to be able to handle technical issues remotely [23]. Therefore, an attacker can either exploit this already installed access or install a backdoor via a network connection to the target system whereas usually, files are to be modified. The authors of [10] argue that in the installation phase the chances are the highest to detect a backdoor as future access by the attackers is often obfuscated.

Thetick - A simple embedded Linux backdoor

Thetick was created by an internet security company called NCCGroup. In their GitHub account, they published the malware [48] together with a short installation instruction. Interestingly, they report that the backdoor includes a command and control console whereas the command and control structure is also an inherent part of botnets.

2.1.2 Botnet

The authors of a paper in which a survey on botnets is conducted[16] a Botnet is defined as follows: "Network of compromised computers called *Bots* under the remote control of a human operator called *Botmaster*." Further, they describe the installation of such a Botnet as follows: Typically a script is executed to download the bot binary mostly from a bot that initially exploited the victim's system. Then the binary is installed on the machine. To successfully join the botnet the system needs to contact the botmaster's server. As the botmaster usually tries to hide his IP address behind a DNS name the system first needs to find out its DNS name by connecting to a DNS server. Finally, the system can connect to the botnet which consists of three authentication steps: First, the bot needs to send a PASS message to authenticate itself to the server. Then, the botmaster asks the bot to provide a password that is contained in the bot binary. As the last step, the botmaster is required to provide a password to the bot. Hence it can be prevented that an unfamiliar botmaster can take control of a bot.

After an installation as described above a botmaster can attack systems with its swarm. With the help of its bots, he/she can execute e.g. Distributed Denial of Service (DDoS) attacks. These attacks can e.g. lead to economical damages as it potentially overloads a web server which can result in an inaccessible company/customer website.

Bashlite vs. Mirai - Malware programs designed for DDoS attacks

The Bashlite malware and its derivative Mirai "infect IoT devices (e.g., broadband modems and surveillance cameras) accessible with vulnerable, known authentication credentials" [45]. It can consist of multiple command and control servers. In contrast to Mirai, the IPs of Bashlites' servers are static and don't make use of obfuscation through DNS. Another advantage of Mirai is that all versions of it come with a built-in scanner which allows bots to search for vulnerable devices whereas only some types of Bashlite malware provide this feature as an extension. Both malware programs prevent the bots from being added to other botnets by disabling their communication capabilities.

2.1.3 Ransomware

In general, Ransomware is divided into two types: One blocks access to files on the attacked system by encrypting them and the other locks the system such that the victim cannot log into it. The way of its distribution is often through phishing as indicated by the authors of [41]. Looking at the setup of advanced ransomware attacks sometimes includes techniques to prevent a user from e.g. restoring or rebooting his/her system or hiding the attack from the user. For a successful blocking of the access of files respectively of the system, the malware must induce a connection between the victim's system and the attack system to receive the encryption key. Whereas by now the system would already be locked the encryption of the file ransomware could be extended by searching for important files or renaming and/or relocating files before the encryption. Often a mechanism is established through which a victim can restore his files. Depending on the encryption technique which can be asymmetric, symmetric, or hybrid (a combination of symmetric and asymmetric) the decryption key to the encrypted files is handed out when the ransom has been paid.

The ransomware attack can be trickier to execute as long as the IoT device under attack doesn't include a display where the attack message can be shown to the victim. Before installing malware an attacker first needs to find a device that is controlled by a human operator. Therefore, the attacker needs to have or gain information about the topology of the network the IoT device is a part of [26]. On one hand, such an attack is more complex than an attack on systems that have displays but on the other hand, the impact of an attack on IoT devices results often in higher damages because of its connectivity to multiple devices. Due to the importance of such systems ransoms could potentially be set higher than in traditional attacks. All in all, as stated by the authors of [26], the number of ransomware attacks on IoT devices has been marginal so far due to its complexity but attackers who have chosen ransomware attacks on IoT devices would mostly target systems of high importance e.g. smart railway systems or cars, which retains the defence of such attacks essential.

RansomwarePoC - Extending educational purpose ransomware

The RansomwarePoC malware is a ransomware that encrypts files on a system. The RansomwarePoC [29] malware extends the simple CriptSky [11] malware by adding the RSA encryption which is an asymmetric encryption mechanism. The public key with which the file is encrypted is provided to the victim whereas the private key is stored in the command and control server of the attacker. When the attacker gets the ransom he can send the private key to the user who then can use it to decrypt the encrypted files.

2.1.4 Rootkit

According to the author of [69] a rootkit is "some kind of hacking toolset (sniffers, network scanners etc.) combined with one of a number of Trojan horses". The latter are programs that try to impersonate normal programs such that the victim respectively its anti-malware program doesn't find them. Once a rootkit is installed, attacks are performed via automated tasks or are hidden behind UNIX utilities. By producing unnecessary processes (opening TCP/UDP ports, registry keys, and stored files) and disabling malware file access attackers try to prevent the detection by anti-virus software. Finally, attackers might try to hinder the deletion of the programs.

Bdvl (bedevil)

The Bdvl malware is a rootkit malware but it also comes with backdoor functionalities [15]. As introduced before it has a wide range of functionalities such as keylogging, stealing files, and even passwords whereas it operates in the user space it tries to "intercept calls from binaries to libraries" to load the malware's functionalities into the libraries [9]. Further, the malware includes a function that hides files and processes that stem from the rootkit. The Bdvl rootkit offers additional functionalities such as hiding ports and the logging of system logins that occur through an ssh connection.

2.1.5 Further IoT Malware/Attack Types

There exist further malware types that are not found in the database and that therefore haven't been used to classify malware in this thesis. Still, an incomplete selection is shortly discussed in the following sub-sections as they can have an impact on IoT network attacks.

Virus

Viruses are programs that can spread across different systems in a network [65]. Continuing, they "can be used to steal information, harm the host system and build botnets". Further, in comparison to worms, viruses would need some human involvement to be

replicated. As human involvement is limited in the IoT world it could be argued that worms pose a bigger threat than viruses in this area.

Worms

Similarly, to viruses worms spread themselves without the active involvement of attackers. In difference, they try to find weaknesses of a system to propagate themselves and don't rely on the help of the systems' user [65]. In addition to that, worms would be often used to slow down the processes of a system or to damage it. Therefore, they are especially dangerous for IoT devices due to their already limited resources and their high level of connectivity to other IoT devices.

Hacking tools: Network Scanner, Spyware and Keylogger

Sniffers can be further divided into network scanners and spyware programs. Whereas spyware mostly focuses on the users' activities keyloggers specialise in the user's keyboard activity [65]. Further, network scanners focus on listening to the network activity of a system. Especially, the latter sniffer type can be crucial for an attacker to investigate the IoT topology. As already stated above, sniffers can come in combination with a rootkit.

Zero-day exploits

Especially dangerous for IoT devices are zero-day exploits as most of these devices only consist of only a handful of software due its limited resources. Therefore, if one of these softwares have a bug that can be exploited by attackers often a major part of the system's use case or functionality could be broken. As such an attack is often custom to a specific software vulnerability it seems to be hard to differentiate such an attack from a benign system's behavior.

2.2 System Calls

A system call is a request from the user space to the kernel which has control over the system [8]. Further, it is indicated that the amount of system calls is limited by the size of the systems' sys_call_table which usually would be 256. In Unix systems, for each system call exists a wrapper routine in the C library which a programmer can use to perform a system call and by that access the CPU, memory, and disks in the kernel space.

The authors of [56] outline that when running a program that copies data from one file to a new file several system calls are performed in the background (see below). Also, each system call could be grouped into six different categories: Process control, file management, device management, information management, communication, and protection. Table 2.2 shows an overview of the tasks of the system calls depending on their categories

combined with some examples of system calls. Further, it is pointed out by the authors of [56] that sometimes the names of system calls of file and device management have a big similarity. Some operating systems such as UNIX would have even chosen to combine system calls that stem from different categories into uniquely named system calls.

2.2.1 Example of system call steps for a simple task

In the following ordered list, we find the system calls needed to copy data from one file to a new file. Depending on the success of the copying process between around eight and 15 system calls (although a loop of reading & writing text lines could produce multiple system calls) need to be executed. The system calls were described and generalized by the authors of [56] such that it can be understood independently from the underlying system. Further, depending on the system, there could be additionally required system calls.

1. Acquire filename of input file which will be copied
2. Write prompt to screen (of the input file)
3. Accept input respectively input path
4. Acquire the filename of the destination file
5. Write prompt to screen (of destination file)
6. Accept destination filename, respectively its path
7. Open input file
8. If the input file does not exist, abort
9. Create destination file
10. If the destination file already exists, abort
11. Read from input file
12. Write to the destination file (loop with 11. until read fails)
13. Close destination file
14. Write a completion message to the screen
15. Terminate process

	Example of UNIX system calls	Tasks of system calls
Process Control	fork(), exec(), exit(), wait()	System calls that handle the (abnormal) termination of processes, creation of new processes or the control of their execution
File Management	open(), read(), write(), close(), getxattr(), setxattr()	- Creation (open()) and deletion of files or folders - Read, write, and close a file/folder - Getting and setting of file attributes (e.g. type or name) - Some systems include system calls for copying or moving files/folders, other systems implement these as APIs consisting of different system calls and code
Device Management	ioctl(), read(), write()	Managing resources/devices (e.g. main memory, disk drives, and access to files) by granting or temporarily prohibiting control until enough resources are free. In the case of a multi-user system, this would include requesting a resource for exclusive use and releasing it when done.
Information maintenance	getpid(), alarm(), sleep()	Returning information from the operating system to the user program. The information can reach from showing the time or date to printing information about the memory and disk space etc. Further system calls provide dumping memory e.g. in case of abnormal termination of a process for debugging purposes.
Communication	pipe(), shmget(), mmap()	**Message-passing model**: Opening and accepting connections, internal system communication between processes, or inter-system communication via a network. **Shared-memory model**: Creation and allocation of memory owned by other processes. This requires that processes agree to remove the memory access restrictions of the operating system.
Protection	chmod(), umask(), chown()	Restricts access to the resources that a system provides. Manipulation of access restrictions can be achieved through system calls that set and get permissions for files or folders. Further, access for different kinds of users can be managed (allow/deny user).

Table 2.2: System calls overview based on information from [56]

2.2.2 System Tracing

For Linux systems, the linux-tools-common package includes the perf program which enables the system call collection by the perf-trace command (see command documentation [33]). Figure 2.1 shows the output when executing *perf trace -S -T -a* on a Raspberry Pi that has been performed to retrieve system calls for the underlying dataset. While the first entry on each line shows the uptime of the system in milliseconds the time in the brackets is the time that shows how long the system call was executed. The system call is shown between program id (PID) and a bracket consisting of system call function parameters. The last line entry after the equal operator is the returned value. While a return value of -1 can indicate that an error occurred a 0 can stand for a successful operation.

```
ubuntu@raspberrypilayer:~$ sudo perf trace -S -T -a
         ? (         ): qemu-system-aa/2205  ... [continued]: ppoll())                                                = 1
132213.095 ( 0.015 ms): qemu-system-aa/2205 read(fd: 107<anon_inode:[eventfd]>, buf: 0xffffff6229c8, count: 512)  = 8
132213.197 ( 0.313 ms): qemu-system-aa/2205 io_submit(ctx_id: 281472665899008, nr: 1, iocbpp: 0xffffff620a68)     = 1
132213.556 ( 3.448 ms): qemu-system-aa/2205 ppoll(ufds: 0xaaaadfb0e880, nfds: 81, tsp: 0xffffff622cc8, sigsetsize: 8) = 1
132217.062 ( 0.010 ms): qemu-system-aa/2205 read(fd: 20<anon_inode:[eventfd]>, buf: 0xffffff6229c8, count: 512)   = 8
132217.109 ( 0.034 ms): qemu-system-aa/2205 write(fd: 104<anon_inode:[eventfd]>, buf: 0xaaaab1b03090, count: 8)   = 8
         ? (         ): dockerd/1117  ... [continued]: epoll_pwait())                                             = 0
132217.178 (         ): qemu-system-aa/2205 ppoll(ufds: 0xaaaadfb0e880, nfds: 81, tsp: 0xffffff622cc8, sigsetsize: 8) ...
132226.663 ( 0.007 ms): dockerd/1117 epoll_pwait(epfd: 5<anon_inode:[eventpoll]>, events: 0xffff67ffdd88, maxevents: 128, sigsetsize: 1
) = 0
         ? (         ): dockerd/1020  ... [continued]: futex())                                                   = -1 ETIMEDOUT (Connection t
imed out)
132226.680 (         ): dockerd/1117 epoll_pwait(epfd: 5<anon_inode:[eventpoll]>, events: 0xffff67ffdd88, maxevents: 128, timeout: 1, s
igsetsize: 187651072573749) ...
         ? (         ): dockerd/1454  ... [continued]: futex())                                                   = 0
132227.228 (         ): dockerd/1020 nanosleep(rqtp: 0xffff7f7cc6c8)                                          ...
132228.025 ( 0.005 ms): dockerd/1454 epoll_pwait(epfd: 5<anon_inode:[eventpoll]>, events: 0xffff64ff7d88, maxevents: 128, sigsetsize: 1
) = 0
132226.680 ( 1.276 ms): dockerd/1117  ... [continued]: epoll_pwait())                                             = 0
132228.038 (         ): dockerd/1454 epoll_pwait(epfd: 5<anon_inode:[eventpoll]>, events: 0xffff64ff7d88, maxevents: 128, timeout: 99,
sigsetsize: 187651072573749) ...
132227.969 ( 0.004 ms): dockerd/1117 epoll_pwait(epfd: 5<anon_inode:[eventpoll]>, events: 0xffff67ffdd88, maxevents: 128, sigsetsize: 1
) = 0
132227.991 ( 0.014 ms): dockerd/1117 futex(uaddr: 0x4000ffe150, op: WAKE|PRIVATE_FLAG, val: 1)                   = 1
132228.032 ( 0.004 ms): dockerd/1117 epoll_pwait(epfd: 5<anon_inode:[eventpoll]>, events: 0xffff67ffdd88, maxevents: 128, sigsetsize: 1
) = 0
132217.178 (12.329 ms): qemu-system-aa/2205  ... [continued]: ppoll())                                           = 0 (Timeout)
132228.043 (         ): dockerd/1117 futex(uaddr: 0x4000105550, op: WAIT|PRIVATE_FLAG)                        ...
132229.575 ( 0.027 ms): qemu-system-aa/2205 write(fd: 33<socket:[33433]>, buf: 0xaaaadf67e150, count: 16)         = 16
132229.628 ( 0.131 ms): qemu-system-aa/2205 ppoll(ufds: 0xaaaadfb0e880, nfds: 81, tsp: 0xffffff622cc8, sigsetsize: 8) = 0 (Timeout)
         ? (         ): SPICE Worker/2430  ... [continued]: ppoll())                                             = 1
132229.786 (         ): qemu-system-aa/2205 ppoll(ufds: 0xaaaadfb0e880, nfds: 81, tsp: 0xffffff622cc8, sigsetsize: 8) ...
132229.929 ( 0.009 ms): SPICE Worker/2430 write(fd: 34<anon_inode:[eventfd]>, buf: 0xffff227fad10, count: 8)     = 8
132229.944 ( 0.008 ms): SPICE Worker/2430 ppoll(ufds: 0xffff227facd0, nfds: 1, tsp: 0xffff227fac68)              = 1
132229.956 ( 0.019 ms): SPICE Worker/2430 read(fd: 21<socket:[33432]>, buf: 0xffff227facd8, count: 16)           = 16
132229.980 ( 0.008 ms): SPICE Worker/2430 ppoll(ufds: 0xffff227facd0, nfds: 1, tsp: 0xffff227fac68)              = 0 (Timeout)
```

Figure 2.1: Screenshot from system tracing output

2.3 Preprocessing Techniques

The most widely used preprocessing techniques, especially in Host-Based Detection Systems, are also used in Natural Language Processing [44]. In the following sub-sections, some NLP preprocessing techniques will be outlined.

2.3.1 N-Grams/Bag-of-Words

Whereas single words are referred to as Unigrams (n=1) in the realm of n-grams, higher numbered n-grams are a bag-of-words representation consisting of multiple consecutive (except skip-grams see subsection 2.3.1) words or letters. The n-gram approach increases

the amount of the input dimension of a model in comparison to single words (Unigrams) depending on the potential number of combinations of words. According to the authors of [67] the feature space could therefore grow exponentially with L^n where L is the number of unique system calls. For a dataset of 200 distinct system calls the 4-gram could theoretically produce up to 1.6 billion different n-grams.

Sliding windows

Sliding windows is a preprocessing technique where a special focus is put on the context of words. The size of the window is mostly odd as there is a focus word in the middle and an amount x which represents the distance from the focus word to the first respectively the last word in the BOW. This results in a total window size of 2*x + 1 [20]. This approach helps encode the meaning of a word depending on its context. A word can have different meanings depending on its context (e.g. "type").

Skip-Grams

Another way of refining the N-Gram approach is by creating n-grams where not the words which are directly after each other are put into a bag-of-word representation but by ignoring one or more consecutive words in a sequence. On one hand, following performing this technique information about neighboring system calls gets lost. On the other hand, longer contexts can be taken into account.

2.3.2 One-Hot-Encoding

Words need to be converted into numeric values or vectors as most machine learning algorithms require numbers as input. As words or letters can occur in nearly unlimited combinations they don't have an absolute ordering. Therefore, a meaningful ordering could be achieved by e.g. taking the number of occurrences of each system call in a specific sequence as input or their relative amounts of occurrences in a sequence with respect to the number of occurrences in the other sequences with the help of Term Frequency-Inverse Document Frequency (see section 2.3.4). Another way of encoding yields the one-hot-encoding by making a binary vector representation of a word or an n-gram. Per word or n-gram a vector with length L is created which has L-1 zeros and a single one. The one represents the position of the word or the letter in the input dictionary. Therefore, the first one-hot-encoding of a word within a vocabulary with length five would be [1,0,0,0,0] and the last word would be represented by [0,0,0,0,1]. This approach makes it possible to directly input the words or letters into a model where then the model needs to interpret the order of the words.

2.3.3 Term Frequency

To calculate more than one term frequency per word the dataset needs to be split into different sequences. E.g. books are already naturally split into chapters that could be used as sequences whereas chapters themselves could be split based on sentences. When being served with such split sequences one can either count the occurrences of each word per sequence or calculate the relative term frequency by taking the term frequency of a sequence and dividing it by the term frequency of the word over all sequences. The following TF-formula is also used to compute the Term Frequency-Inverse Document Frequency (TF-IDF) which is explained in the next section (see 2.3.4).

$$TF = \frac{TermFrequencyInSequence}{TermFrequencyOverAllSequences}$$

2.3.4 Term Frequency-Inverse Document Frequency

The TF-IDF-formula expands the relative term frequency approach and multiplies it with the inverse document frequency (IDF) which takes into account how many documents/sequences a term occurs. The IDF is the logarithm of the number of documents (here called sequences) divided by the number of sequences a term is occurring.

$$\text{TF-IDF} = TF \cdot log(\frac{NumberOfSequences}{NumberOfSequencesWithTermOccurrence})$$

Therefore, a word that occurs in every sequence results in a TF-IDF value of zero (log(1)=0) whereas the lower the number of document occurrences of a word, keeping the term frequency constant, the higher the TF-IDF value.

2.3.5 Odds ratio (OR)

The odds ratio preprocessing methods' "basic idea is that the distribution of features on the relevant documents is different from the distribution of features on the nonrelevant documents" [70]. The Odds Ratio value of term t and category c_i consists of two dependent probabilities and their complementary probabilities: First, the probability of a term appearing in a document or sequence if it is a member of the category c_i ($P(t|c_i)$) and second, the probability of t occurring if it is not part of c_i ($P(t|!c_i)$). Therefore, the complementary probabilities (1-$P(t|c_i)$ and (1-$P(t|!c_i)$)) are the probabilities of a term not appearing in a document or sequence if it is a member of a category respectively the term not occurring in a sequence if it is not part of a specific category. It is defined by the following equation:

$$OR(t, c_i) = log(\frac{P(t|c_i)(1 - P(t|!c_i))}{P(t|!c_i)(1 - P(t|c_i))})$$

2.3.6 Positional Encoding

The positional encoding is a special preprocessing technique for the Transformer model to preserve relative and absolute position information of embeddings [62]. The founders of the model achieve it by adding the positional encodings to the embeddings. The positional encodings (PE) are calculated through the following sinus and cosinus functions where i is the index of the term in the dictionary, *pos* its absolute position in a sequence, and *dmodel* the size of the input dictionary. Therefore, a word in a dictionary that has an odd index is calculated by the cosinus and a word with an even index is positioned by the sinus function.

$$PE(pos, 2i) = sin(pos/10000^{2i/dmodel})$$

$$PE(pos, 2i+1) = cos(pos/10000^{2i/dmodel})$$

2.4 AI Models

In the following sections, different artificial intelligence models are explained. Starting with the standard neural network and ending with deep learning models such as Transformers or Convolutional Neural Networks. Not all models are implemented in this thesis but are at least used in related work.

2.4.1 Neural Network

The author of [63] defined the structure of a neural network as follows: "An artificial neural network (or simply neural network) consists of an input layer of neurons (or nodes, units), one or two (or even three) hidden layers of neurons, and a final layer of output neurons." Figure 2.2 illustrates an example of a neural network. The lines between the nodes show the weights of each layer. The input of a node a_i of the hidden layer is calculated by the input node x_1 and the weights $w_{1,x}$ which is shown in the following formula whereas D is the height dimension of the current layer (D=4 in the input layer of this example):

$$a_1 = \sum_{d=1}^{D} x_d \cdot w_{1,d}$$

2.4.2 Support-Vector-Machine (SVM)

A Support-Vector-Machine is often used to classify data points in two classes in a supervised fashion. The datapoints are often represented on an x-y-scale and the datapoints stemming from two distinct classes are then tried to be separated by a so-called decision

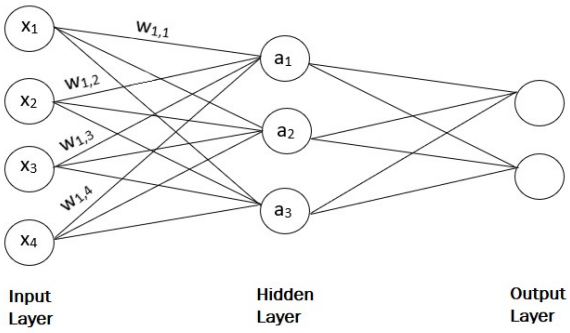

Figure 2.2: Figure of a(n) (Artificial) Neural Network based on the figure of the author of [63]

boundary which is constructed by a hyper-plane [18]. Further, the SVM can also be used as multi-class classifier when repeating the hyperplane search multiple times whereas for each class the binary classification of an SVM is performed where the datapoints of the one class are compared against all other classes' datapoints that are put together into one class.

2.4.3 Random Forest

Random Forest is another form of a rather simple decision model which consists of different so-called decision trees. A decision tree mostly consists of multiple decisions where while training each decision step is assigned a comparison value of a feature x e.g. $x < 2$. If x is either smaller or greater equal two the corresponding path is chosen. According to the authors of [18], at the end of such a tree consisting of multiple decisions, the input is then assigned to a class. Further, they state that a random forest is a combination of multiple decision trees whereas normally, they are implemented in that each tree gets a vote for one class and then the class with the highest amount of votes is chosen.

2.4.4 Recurrent Neural Network

Recurrent Neural Networks (RNNs) are neural networks that consist of interlinked neural networks. It can be used to model sequences of words whereas each word is represented by a network whereas the last network combines the results of every network in a sequence [20]. The networks are interconnected through an additional input layer which was coined *state* by the author of [14]. In Figure 2.3 a representation of an RNN over a sequence with length four is shown. The θ-symbol represents the shared parameters over the whole connected network [20]. However, the recurrence enables the network to learn sequence information the connectivity often produces vanishing gradients which possibly neglects

the information of the first networks of a sequence as indicated by the authors of [50] and therefore words placed at the beginning of a specific sequence have little influence on the sequence network. As long-term information of a sequence gets lost a new model was found to address this problem. The so-called Long Short-Term Memory (LSTM) model is an adaptation of the RNN. It will be discussed in the next section.

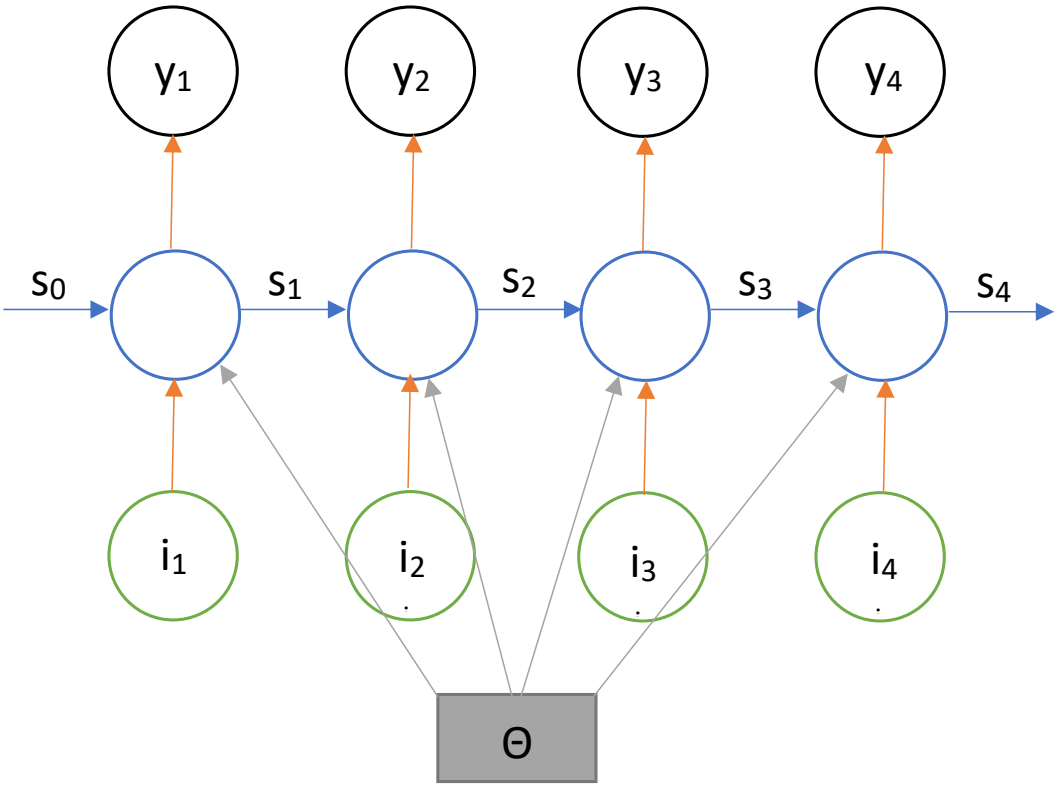

Figure 2.3: RNN shown in an unrolled fashion based on the figure in the book [20]

2.4.5 Long Short-Term Memory (LSTM) model

The idea of the LSTM model is based on knowledge about the human brain. The brain memorizes new information for a short period of a few seconds [19], and only if the new information is important enough it will be memorized for a longer time. This idea is

captured in the so-called "gates" of an LSTM cell (depicted in Figure 2.4). There are three types of gates: input, forget and output gate. The LSTM model is a recurrent neural network that consists of multiple LSTM cells.

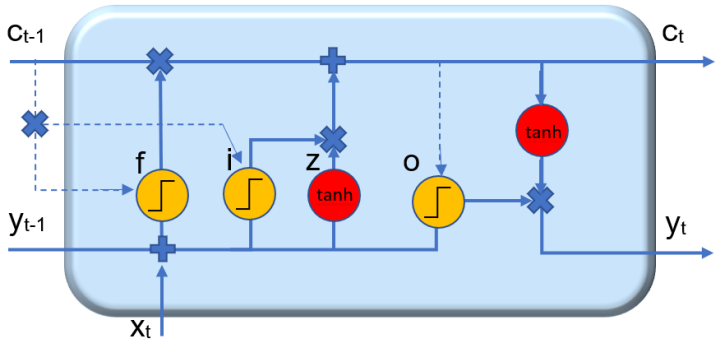

Figure 2.4: Depiction of an LSTM cell based on a figure of the authors of [61]

First, we focus on the input step and its activation function (see black arrows in Figure 2.5): The input gate (see i in Figure 2.4) decides on how much of the input should be contained in the memory cell at the end. According to the authors of [61], the inputs of the input gate are updated in this step which is shown in the following equation where the current input x_t is combined with the output of the last LSTM cell y_t and the value of the last cell c_{t-1}. Their parameters W_i respectively w_i are their weight matrices respectively c_{t-1}'s weight vector and b_i is the bias vector:

$$i_t = sigmoid(W_i x_t + W_i y_{t-1} + w_i \odot c_{t-1} + b_i)$$

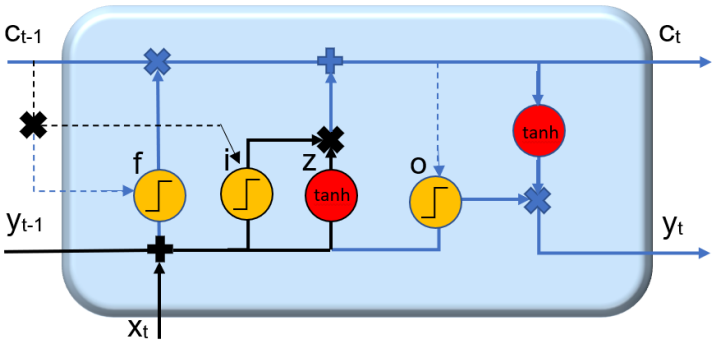

Figure 2.5: Depiction of an LSTM cell based on a figure of the authors of [61] with focus on input gate i_t and its activation function z_t

The input gate i_t is then point-wise multiplied with the result of the activation function z_t in which the weights of the vectors x_t and y_{t-1} and the vectors itself are summed up with the bias vector:

$$z_t = tanh(W_z x_t + W_z y_{t-1} + b_z)$$

In the forget step respectively the forget gate, f_t decides how much information of the previous cell should be kept by taking the same input (x_t, y_t and c_{t-1}) as in the input step for the sigmoid function in equation i_t but with different weights and without activation function z_t (see Illustration 2.6). By adding the forget gate and the input gate which are multiplied with the previous cell state respectively with the block input z_t we receive the cell state c_t of the current cell:

$$c_t = z_t \odot i_t + c_{t-1} \odot f_t$$

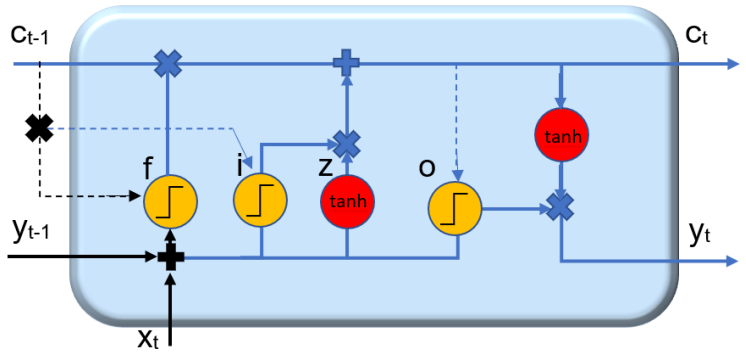

Figure 2.6: Depiction of an LSTM cell based on a figure of the authors of [61] with focus on the forget gate f_t

The output gate o_t again combines the cell inputs and puts them into a sigmoid function with their weights. The result of this is then multiplied with the activation of the cell state which results in the output y_t of the cell (see the following equation):

$$y_t = tanh(c_t) \odot o_t$$

2.4.6 Deep Belief Network

As standard neural networks also struggle with vanishing gradients when having multiple hidden layers Deep Belief Networks (DBN) have been constructed that consist of so-called Restricted Boltzmann machines (RBMs). Summarized RBMs are 2-layer networks where the nodes have a probabilistic dependency, although the dependency is restricted to take place between nodes of distinct layers [25]. The values of the nodes are learned in an unsupervised fashion to be either 0 or 1 depending on these probabilities. It is reported that in this way the learning can be done more efficiently. By stacking RBMs one receives a Deep Belief Network. Enabling a DBN to classify data based on labels the output of a DBN must be extended with a suitable classifier such as e.g. softmax.

2.4.7 Autoencoder

An Autoencoder network can be described as being a combination of two neural networks where the output layer of the encoder network is used as the input layer of the decoder network. It can be used to reduce the dimensionality of given data e.g. by ignoring insignificant pixels of a photo, by encoding the data into a feature representation and after that decoding it into a data representation with lower complexity [64]. Further, Autoencoders can be used to separate unlabeled data into different groups but apparently, they can also be used to classify labeled data while finding "the latent variables that explain the input" [21].

2.4.8 Convolutional Neural Network

Like in the basic neural network a convolutional neural network (CNN) consists of different layers that perform matrix multiplications. Contrary, to the basic neural network so-called convolutions are performed during these multiplications. Additionally, a CNN mostly consists of multiple hidden layers where the convolutions are performed. A convolution is carried out by doing multiple multiplications on a matrix with the help of a *filter* that is chosen to be smaller than the original matrix to enable the multiple multiplications. The term convolution describes the calculation of one matrix entry of the next layer that consists of an aggregated value based on calculations on multiple entries of the previous layer [2]. Controlling the output size of a given layer is key to receiving a working model as at the end of the network the current layer must be mapped to a linear layer. The output size of a convolutional layer can be influenced by different factors, namely filter size (F), amount of filters (K), stride (S), and the amount of zero paddings (P) but also depends on the dimensions of the input matrix. For a 2-dimensional matrix with width W_1 and height H_1 the resulting matrix receives the W_2 and height H_2 according to the authors of [12] as follows:

$$H_2 = \lfloor \frac{H_1 F + 2 \cdot P}{S} \rfloor + 1$$

$$W_2 = \lfloor \frac{W_1 F + 2 \cdot P}{S} \rfloor + 1$$

Stride

The stride is the step size of the filter moving over the matrix doing the calculations. Figure 2.7 depicts this movement when having a stride of two and a matrix of dimension 10x5 and a filter of dimension 3x3. Two observations can be done: First, not only the movement of the filter on the horizontal line surpasses one dimension but also the movement in the vertical direction. Secondly, the last column of the matrix is not considered with the given dimension. When using the above formulas the resulting height of the next layer matrix would be 2 whereas the width of it would be 4.

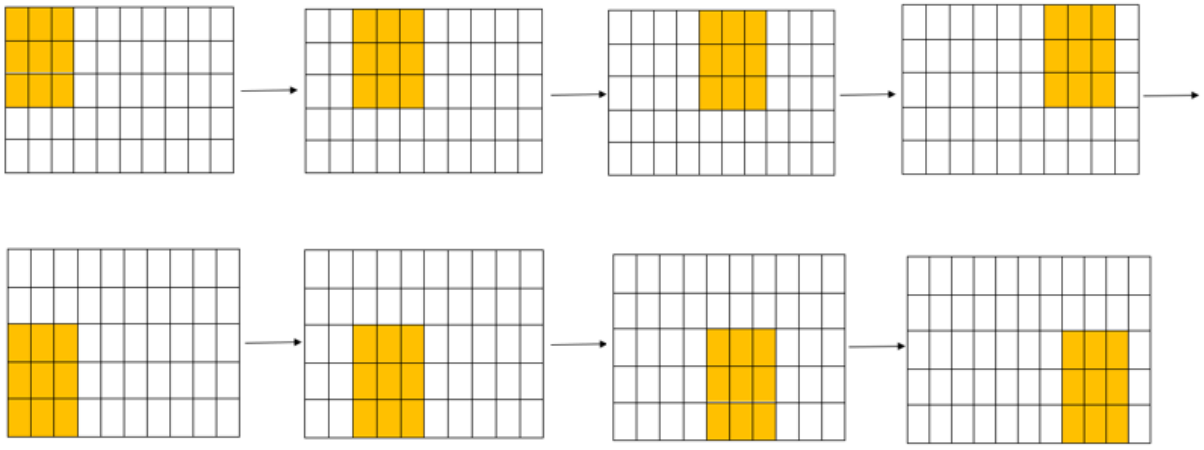

Figure 2.7: Convolution on 10x5 matrix with stride = 2 and kernel dimension 3x3

Zero-Padding

Zero-Padding or Padding can help prevent that a column of a matrix is ignored during a convolution as in the prior example. Padding of length 1 sets a frame consisting of entries that are set to zero around a matrix before the convolution happens (see Figure 2.8). This can also become necessary when the goal is to keep the same dimensions or even to increase the dimension size of the resulting matrix [12].

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	0
0	7	8	9	10	11	12	0
0	13	14	15	16	17	18	0
0	19	20	21	22	23	24	0
0	25	26	27	28	29	30	0
0	31	32	33	34	35	36	0
0	0	0	0	0	0	0	0

Figure 2.8: Matrix of dimension 8x8 without (left) and with padding of 1 (right)

Pooling

Whereas a normal convolution takes aggregate values max-pooling e.g. is done by choosing the biggest value in the given filter region. This pooling strategy augments the importance of entries with a high value in the original matrix. Therefore, in image classification e.g. this strategy lays importance on more visible/darker pixels. Further, average pooling yields another pooling strategy by taking the average of all values being looked at by the filter during one step of convolution.

2.4.9 Transformer

In comparison to the LSTM model the Transformer model does not rely on a Recurrent Neural Network but puts attention on the most important parts of a specific sequence [62]. Further, the model lies importance on three different types of (self-)attention are implemented in the original model of the authors of [62]. These are briefly outlined below:

1. Self-attention in the encoder layers: Self-Attention is the attention of a word put on every other word in a sequence and itself respectively its positional encodings of a previous encoder layer.
2. Self-attention in the decoder layers: Similarly the decoder positions put attention on its position and the other words' positions. In comparison to the encoder following words of a sequence need to be masked such that the decoder not directly learns the order of the words in the output language.
3. Encoder-Decoder attention: For each word in an output sequence with the corresponding input sequence its attention concerning all the words of the input sequence is computed.

Attention in this model is calculated by computing the activation with softmax of the keys and queries respectively their weights (Q and K) which are scaled by the root of the key dimension (see following equation [62]). Further, the embedding vectors of the words of a sequence are put together into a matrix. For each word the matrix is used to be calculated with the already mentioned query and key (and value) layers which result in the matrices Q and K (and V). As V is the matrix that is not multiplied by another matrix inside the softmax whereas Q and K are not. As the correct or the most important feature is not necessarily learned automatically this is carried out multiple times in parallel. This process is called multi-head attention [62].

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

2.5 Dataset

The underlying dataset for this thesis stems from the author of [58] and was published on the IEEE Dataport website [59]. According to the author of [59] the dataset "models the internal behavior of an IoT spectrum sensor" constructed with a Raspberry Pi that has been connected to the ElectroSense platform while the devices' system calls have been recorded during attacks respectively no attack. The attacks can be categorized into four different malware types while some of them were also sub-categorized into different attack phases, e.g. the system's behaviour when the Bashlite malware is or has been installed is split into four sub-phases, namely the installation phase and into three different attack respectively flooding techniques (UDP, TCP and HOLD). Further details on the attack phases and the dataset are discussed in the *Data Exploration* sub-section in chapter 4. In the following section, some characteristics of the ElectroSense platform and the connected Raspberry Pi from which the system calls were collected are presented.

2.6 ElectroSense - Radio Frequency Platform

The ElectroSense crowdsourcing platform was established to monitor electromagnetic waves that could e.g. stem from cognitive radios or electronic devices [52]. An original unit for collecting radio frequency data would consist of a Raspberry Pi as already indicated in combination with a radio frequency front-end, a radio antenna, a GPS module, and a radio signal receiver called RTL-SDR. The Raspberry Pi used in the creation of the dataset serving as a basis for this thesis was "a Raspberry Pi 4 with an ARMv7 rev 3 processor and 4 GB of RAM" [58]. While the IoT device was an active part of the ElectroSense platform different malware types were installed and attacks were performed on it. In the meantime, the behavior of the system was recorded by collecting system call traces. Further, the collection of system calls from the Raspberry Pi would happen while it receives radio frequency data and sends it to the ElectroSense platform [58]. As the ElectroSene platform is an open-accessible network it is a good example of potential dangers in the IoT realm.

Chapter 3

Related Work

3.1 Malware Detection through ML models based on System Calls

There seem to be two major strategies to train a model for malware detection in the research community: Whereas some (e.g. [42]) only train their model with malware samples e.g. others (e.g. [28]) additionally take samples of a system that has no malware installed. While in the former case the classifier can't decide if a system was attacked or not with the help of multi-class classification, such a model can still be used to determine which type of malware was installed on a system. In the detection strategy where the benign system state is included anomalies can either be found via binary or multi-class classification. If there would be a perfect multi-class classifier it would render binary classifiers unnecessary. As a perfect model doesn't exist yet we have a look at binary and malware classification separately in the following section. Table 3.1 gives an overview of the stated and additional related work and their classification results.

3.1.1 Binary Classification

Binary malware detection in Linux IoT devices based on system calls has already been conducted [55]. The authors of the conference paper were able to successfully classify IoT Malware (98.7 Accuracy) with the help of an RNN. The author of [17] achieved an F1-Score of 96.4 percent with the SVM model whereas the system calls where preprocessed into term frequencies. Eight different SSDF attacks were carried out on IoT spectrum sensors. The authors of [6] used the Odds Ratio (explained in section 2.3.5) preprocessing technique and reached combined with the Random Forest method an accuracy of 97.3 percent.

Similarly to IoT devices, the system calls of an Android device can also be used to determine if a system is under attack or not. In contrast to the Linux realm, the system calls aren't collected based on the system but from the apps installed on the device. These system calls are called *API calls* and permission logs can be extracted from the APK

files of android apps. Those were used in the following papers to detect malware. The authors of [53] established a Bidirectional Gated Recurrent Unit (BiGRU) model based on a combination of an RNN and a CNN. The so-called DexCRNN_BiGRU yielded together with the preprocessing technique Nearest Neighbor Interpolation an accuracy and F1-score of 95.8 percent. The authors of [66] implemented the LSTM model with four hidden layers to detect anomalies on an android system. As in the previously mentioned DexCRNN_BiGru approach used by the authors of [53] the dataset was only preprocessed by a straight-forward splitting of system calls into equally long sequences. In this case, an accuracy of 93.7 percent was reached. But, also simpler models can get good results in binary classification: The authors of [51] e.g. used a Support Vector Machine based on the system calls which classified benign and malicious apps with an accuracy of 95.75% percent. In combination with permission types, the score was 96.88%. The authors of [68] reached an accuracy score of 96.76% with a much more complicated architecture based on a Deep Belief Network (DBN) and Boltzman Machines. Dynamic and static features were used for that classification. When only using the static features where API calls belong to the system state was classified correctly with an accuracy of 89.03%. The authors of [40] used an SGDclassifier to evaluate if Windows API call sequences are malicious or benign and yielded an F1-score of 95 percent. The binary classification seems to work on different systems with similar success rates.

3.1.2 Categorical Classification

As malware classification in IoT devices based on system calls is a rather new field of research there aren't that many publications about it yet. Still, the following two research papers executed classifications of malware programs within the botnet malware type based on Unix system calls. The authors of [39] used the 2-gram preprocessing technique together with a PCA algorithm and then fed multiple one-class Support-Vector-Machines with it. This approach yielded an F1-score of 98.46 percent on six datasets consisting of five different botnet malware. Using the same technique the authors of [3] reached an accuracy of 100% on a dataset stemming from routers consisting of the Botnet classes MrBlack and Mirai. The Mirai class was additionally split into Mirai versions 1 to 4. 100% was also achieved when using the TF-IDF technique together with 1-gram inputs.

The authors of [28] used API calls of an android system which are related to system calls to evaluate different models. The best model which was Multiple-Detection Naive Bayes combined with the Rete algorithm reached a precision of 88.7 percent on their dataset. The authors of [42] used a Convolutional Neural Network in combination with an LSTM. Particularly, they preprocess their data such that recurring system calls are deleted from the dataset and then they perform a one-hot encoding of the system calls. Finally, ten different malicious malware data were collected and then classified by the model with an accuracy of 89.4 percent and 89.2 percent without the LSTM (with the CNN only). Only one malware called Mikey was not classified at all which accounts for up to 10 percent of the accuracy score. Mikey was often mixed-up with another Trojan called Zusy. All other malware types had an accuracy rate of 96.6 percent or above. The authors of [54] tried a similar approach with several different DL/ML models where the CNN yielded the best result. The windows system call dataset they used included eight different malware types.

With the help of the TF-IDF preprocessing technique, an accuracy of 91.0 percent was achieved. The authors of [60] used network traffic features in combination with system call features, although they indicate that the network-flow features alone would only have a precision of 49.9 percent on malware categorization. With the help of the n-grams of the system calls and the random forest model, they yielded a precision of 83.3 and a recall of 81 percent on an android malware dataset.

Work	Scenario	Feature type	System	Classification Type	Preprocessing Technique	ML model	Malware Types	Performance
[42] (2016)	Malware classification	API Calls	Windows [probably]	Multi-Class	One-Hot-Encoding and Deletion of recurring system calls	CNN+LSTM	Adwares & Trojan (10 different malware)	Accuracy: 89.4%
[60] (2019)	Mobile phone malware	Network flows & API calls	Android phones	Multi-Class	2-Grams	Random Forest	Adware, Ransomware, Scareware & SMS Malware (<43 malware families) from dataset of the authors of [43]	Precision: 83.3%
[54] (2021)	Malware Classification	API calls	Windows	Multi-Class	TF-IDF	**CNN**, Random Forest Classifier, Bernoulli Naïve Bayes (NB), Multinomial NB, Gaussian NB, Decision Tree Classifier, AdaBoost Classifier, Bagging Classifier, KNeighbors Classifier, and MLP Classifier	Adware, Backdoor, Downloader, Dropper, Spyware, Trojan, Virus, and Worm	F1-score: 91.0%
[3] (2017)	Botnet malware classification	System Calls (ftrace)	Linux (Virtualized home router)	Multi-Class	2-Grams & PCA / 1-Gram & TF-IDF	SVM	Botnet (MrBlack & Mirai v1-v4)	Accuracy: 100%
[39] (2020)	Botnet malware classification	System Calls	Linux (MIPS CPU Architecture)	Multi-Class	2-Grams & PCA	SVM	Botnet (5/6 different malware families (depending if Downloader is counted as a family or not)	F1-score: 98.46%
[55] (2020)	Telnet attacks	System Calls	IoTPoT (Emulation of different CPU architectures)	Binary	TF-IDF & 10-Gram	RNN	N.A.	Accuracy: 98.7%
[17]	SSDF attack classification	System Calls	Raspberry Pi	Binary	Term Frequency	**One-Class SVM**/Autoencoder	8 different SSDF attacks	F1-score: 96.4%
[28] (2018)	Android applications	API calls	IoTPoT	Multi-Class	Rete algorithm	MDNBS	Worm, Virus, Trojan and Normal	Precision: 88.7%
[66] (2017)	Mobile phone malware	API Calls	Drebin Dataset [5]	Binary	One-Hot-Encoding	LSTM (multiple amounts of system calls sequences)	Four malware families	Accuracy: 93.7%
[51] (2013)	Mobile phone malware	API Calls (& Permissions)	Android	Binary	*Application-wise One-hot-encoding*: Existence of API call in application marked with 0 or 1	**SVM**, J48 and Bagging	Phishing and Banking-Trojans, Spyware, Bots, Root Exploits, SMS Fraud, Premium Dialers, Fake Installers, etc. (49 Malware Families (unbalanced))	Accuracy: 95.75% (96.88%)
[68] (2016)	Mobile phone malware	API Calls & Permissions (& dynamic features)	Android	Binary	*Application-wise One-hot-encoding*	DBN & Boltzman Machines	N.A. (8.09% of the dataset malicious)	Accuracy: 89.03% (96.76%)
[53] (2020)	Android IoT	APK Files	Android IoT	Binary	Nearest Neighbor Interpolation & Average Pooling	**DexCRNN_BiLSTM**, DexCNN, DexCRNN, DexCRNN_GRU, DexCRNN_LSTM, DexCRNN_BiGRU	N.A.	F1-score: 95.8%
[40] (2018)	Windows Anomaly Detection	API calls	Windows	Binary	SGD classifier	SVM	N.A. (64.7% of the dataset malicious)	Accuracy: 95%
[6] (2014)	Linux Anomaly Detection	System Calls	Linux (VX Heavens Virus Collection)	Binary	Odds-Ratio Method	Naive Bayes, J48, AdaboostMI (J48), IBK-5 and **Random Forest**	N.A.	Accuracy: 97.3%
[13] (2017)	Server-Client System	System Calls	Linux	Binary	Relative Term Frequency	Logistic Regression, SVM, Random Forest, LSTM, Bidirectional-LSTM and **inception-like LSTM**	N.A.	Precision: 92.4%
[22] (2021)	Server-Client System [13]	System Calls	Linux	Binary	Relative Term Frequency	LSTM-Transformer	N.A.	Precision: 92.6%

Table 3.1: Overview over studies on system call & Machine Learning based malware/anomaly detection

3.1.3 Effects of Deep Learning model parameters on classification results

CNN

Whereas the authors of [42] implemented a CNN model with Max-Pooling convolution layers, the authors of [54] made use of Average-Pooling convolution layers. This probably stems from the fact that the authors of the two papers implemented two different preprocessing approaches: While in [42] the One-Hot-Encoding preprocessing technique was used, in [54] the input system call sequences have been restructured with the help of TF-IDF. The same applies to the two different dimensions of the Pooling mechanisms: As the input of TF-IDF sequences are 1-dimensional arrays the pooling layers are also one-dimensional whereas in the One-Hot-Encoding of the authors of [42] the Pooling layer is of dimension two.

LSTM

The authors of [66] report that having longer sequences of system calls doesn't necessarily lead to better classification results. In their approach, where an LSTM model was implemented, the highest accuracy was reached when using 100 system calls per sequence, although the next higher amount of system calls per sequence was 500. Further, they tried out different amounts of hidden layers and hidden neurons. Keeping the other parameters constant four hidden layers and 1000 hidden neurons yielded the best results. For the latter, it showed that a higher number of hidden neurons tends to reach better results at least when looking at the number of neurons increasing until 1000. Although these insights are interesting they can only be seen as a tendency but not as absolute values. The parameters might need to be changed according to the total number of system calls and the classification task. In this case, the total number of system calls the model receives lies at 71'030 with 129 different system calls and it needs to perform binary classification.

Transformer

As the Transformer model is a rather new model it was only used in combination with the LSTM for system call based malware classification. The authors of [22] have chosen the following parameters for their transformer: 5 heads, the number of hidden dimensions equal to 64, and a dropout of 0.1 whereas their input had the size of 310.

3.1.4 Effects of adversarial samples on classification results

[4] elaborated on what happens if an android system call based machine learning model would be provided with manufactured malware samples in the test phase by adding system calls that only occur in the benign programs and add those to the malware samples. With manufactured samples consisting of only one percent benign system calls, the performances

of all of its evaluated models declined drastically (e.g. recall dropped from 98.68% to 50.4% with Linear Regression). Although, the authors have used models with limited complexity such as Linear Regression or Random Forest this attack vector will also be shortly discussed and evaluated with chosen models.

Chapter 4

Implementation Details

In this chapter, first, the data distribution is explored and checked for consistency. Then, the implementation specifics of the preprocessing techniques are discussed whereas a focus is put on the altered TF-IDF preprocessing technique which was named TF-DF and its differences from the TF-IDF technique are stated. Further, a pseudo-code extract is shown to see how this technique was implemented in the thesis. After that, model details and two different classification approaches are presented. In the end, it is explained how some experiments with adversarial attacks were conducted.

4.1 Data Exploration

The provided dataset consists of five different system states including four types of malware affecting the system and one where the system isn't affected by any malware. In all system states the devices' (Raspberry Pi) resources were used for collecting radio frequency data and forwarding it to a server. In the benign system state, no additional processes such as e.g. installations or updates are carried out. Below, the executed processes while carrying out attacks or the malware were installed on the system are shortly outlined based on the work of the authors of [58] and [59]:

1. Bashlite (*Botnet*): The Bashlite dataset starts with the installation of the Bashlite malware which results in the fact that the system can connect itself to a Botnet. After connecting to the botnet, the botmaster executes different kinds of flooding attacks (HOLD, TCP, and UDP) through the bot which are performed separately and consecutively.

2. Bdvl (*Rootkit*): The Bdvl dataset is divided into four attack parts. The first two parts handle the installation of the malware which is divided as follows: The first step consists of an install&&make command and in the second step, the malware is built. After the installation, an ssh login is performed by the rootkit user. Finally, the bdvl-specific attack loop is performed which consists of stealing files and the creation and deletion of directories.

3. RansomwarePoC (*Ransomware*): As the name suggests, the RansomwarePoC malware belongs to the ransomware malware type. The dataset only captures one state, namely the encryption of directories.

4. TheTick (*Backdoor*): TheTick is a Backdoor malware. In the given dataset, two states are differentiated: On one hand, the infection of the system and on the other hand the execution of attacks. During the infection phase, the backdoor is installed on the victims' machine. In the attack phase, files are altered and moved between the system of the attacker and the system of the victim.

Whereas the whole dataset consists of roughly 1 Terabyte of data the classification tasks in this work made use of only around 1 Megabyte (the size of the smallest attack state file) of data per class. The main reason for this is that some attack phases haven't required more data to be completed. Further, results of some experiments indicate that using more data doesn't necessarily improve the performance of a model (see section 5.2).

As a first data exploration step, the different states per malware, e.g. installation of malware weren't considered when differentiating between them. Further, only the first step of each malware infection was explored. For a first impression, the 15 most occurring system calls per malware were picked to show potential differences in the number of system call occurrences per malware. The fractional numbers in Figure 4.1 show the relative amount of system call occurrences per dataset. The Figure shows that the system call *open* only appears in one of the five datasets (Normal). Figure 4.2 takes the data of the beforementioned Figure 4.1 but abolishes the most occurring system and context calls, nanosleep and nanosleep*, to better show the relative differences of system call occurrences between the different system states.

If we additionally remove the outliers nanosleep and nanosleep* we see even more differences between the different amounts of chosen system call types (see Figure 4.2) although, the differences of the chosen system calls between Bdvl, Normal and Thetick seem to be minimal.

In a later stage, it was also differentiated between different kinds of attacks. There the focus lies on the distribution of different kinds of system calls. They have been split into three observed groups: First, regular system calls that were executed and ended without being interrupted by other processes. Secondly, context system calls are system calls that are interrupted by another process before they are completed. The context system calls are marked with an asterisk (e.g. nanosleep*) in this work. Lastly, the system calls are irregular as their start time is not in the correct order.

Figure 4.3 shows the distribution of these different types of system calls over the attack datasets (including one benign dataset). It shows that irregular system calls stemming from the perf-trace Unix command don't occur often. Their relative amount to the total number of system calls is invisible to the human eye compared to the other two types. Therefore, they are determined to be insignificant and are not excluded from the dataset. At the same time, context system calls are found to occur on a regular scale. In addition, Figure 4.3 indicates that different malware and attacks have different effects on the number of context system calls. As this evaluation was only done within 10 seconds of data for each

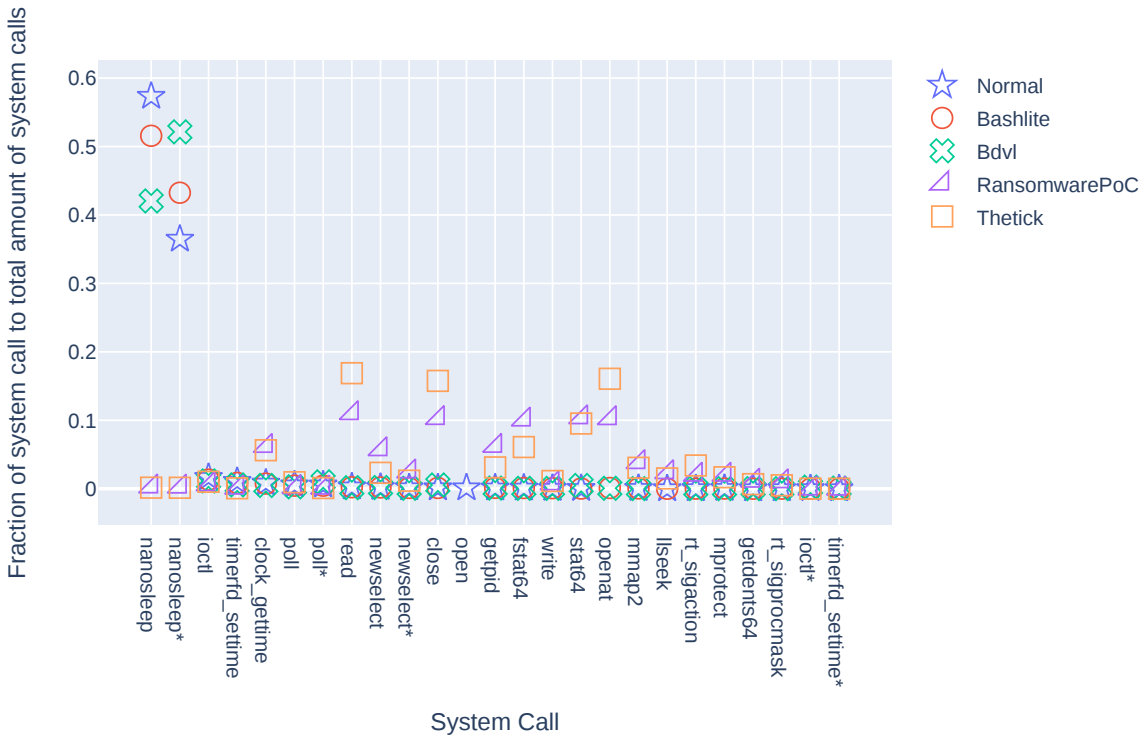

Figure 4.1: Relative Differences Between Top24-Most-Occurring System Calls

attack type next we try to confirm this with an analysis of a bigger dataset. As already stated Figure 4.1 shows the distribution of the 24-most occurring system calls including nanosleep*. This context call makes up 97 percent of all context calls in the analysis over all classes of more than one hour of system calls (per minimum 446'370 system calls per malware) which were randomly selected. Although, the randomness had to be limited to randomly picking sequences in periods of 15 minutes to one hour as some malware hasn't been active any longer (after the activity of the malware the system state can go back to normal again and then cannot be used as evidence for above hypothesis). This shows that the stated hypothesis has to be rejected when comparing the occurrence of nanosleep* in Figure 4.1 with all the context system calls in the smaller dataset in Figure 4.3 as the ranking of the amount of context system calls per malware changed slightly. Where Bashlite has the 3rd place in nanosleep* occurrences it climbed to have the second highest amount of context system calls when looking at a bigger amount of system calls. So the hypothesis that attacks or malware can be solely found by looking at the amount of context system calls has to be rejected because the amount differs depending on how many system calls are studied.

Further data exploration was done to find out if there are system calls that only occur in one attack phase dataset. A first observation was done by looking at 10 seconds of each attack phase (see *NormalBenign10s* in Table 4.4). This yielded nineteen single system or context calls. To find out if these single system calls have a direct connection to the system state half of the datasets were enlarged. The datasets to be increased

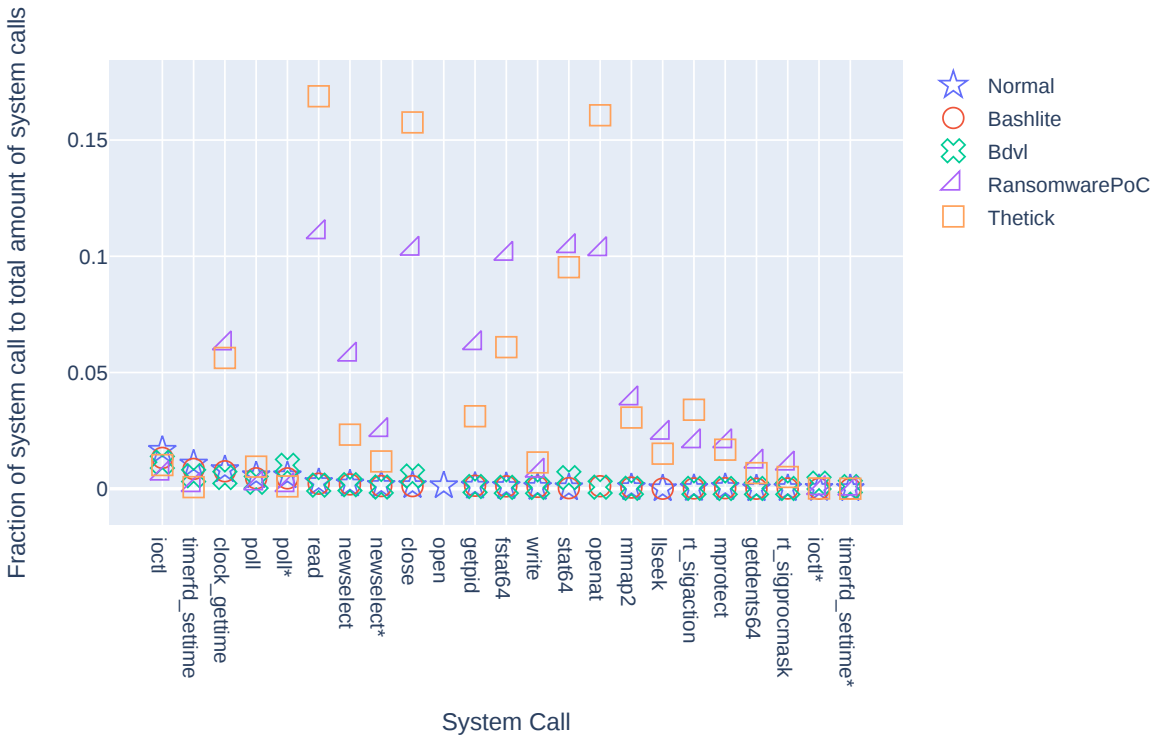

Figure 4.2: Relative Differences Between Top22-Most-Occurring System Calls (nanosleep and nanosleep* removed)

were manually chosen by looking at how long specific attacks endured. For attacks that minimally take one minute, so mostly the attacks that include attack loops, the data size was enlarged to 50 seconds which results in 114'015 system calls per those attacks. This enlargement modified the results of the single system call observation such that the number of system calls that only occur during one specific attack dropped from nineteen to nine (see ***NormalBenign50s*** in Table 4.4). Following these system calls are enlisted:

- RansomwarePoC –> getpgrp & unlinkat
- Bdvl_install&&make –> symlink
- Bdvl_ssh_login –> setgid32
- Normal –> open, open*, lseek, lseek* & getdents*

Still, as in Figures 4.1 and 4.2 we see that the *open* system call has only been executed in the normal dataset. In addition, the normal dataset uniquely consists of the *open* context call, the *lseek* system and context call, and the *getdents* context call. Those system and context calls are mainly used for file management, e.g. getdents is used to get directory entries as stated by the author of [32]. Therefore, these operations could stem from the collection of radio frequency data. Thus, inferring from this the collection of data is

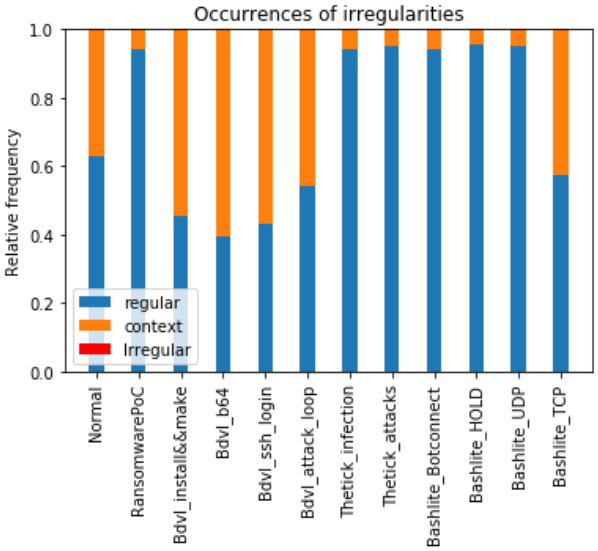

Figure 4.3: Distribution of irregular, context, and regular system calls over different attack types

impeded or interrupted once the malware is installed or attacks are carried out. Next, we look at system calls occurring in the Bdvl malware dataset only. The *symlink* system call creates symbolic links which create a placeholder of a file in a specific directory and from there enable opening the original file which is located in a different directory [36]. The *setgid32* (can handle up to 32-bit IDs) system call sets the group id of the calling process whereas the related system call *getpgrp* has been executed uniquely in the ransomware dataset [34]. The system call returns the process group of the calling process [30]. The other single system call which is also only executed during the Ransomware malware is called *unlinkat* and is used to remove directories [37]. Concluding, the unique system calls of malware datasets are mostly performing advanced file management and process management when comparing them to the benign state where the file management system calls are rather simple.

System Call	Normal	Thetick_attacks	Bashlite_HOLD	Bashlite_UDP
capset	nan	3.0	3.0	3.0
tgkill	nan	1.0	nan	1.0
fadvise64_64	4.0	6.0	6.0	6.0
sendmmsg*	nan	1.0	1.0	1.0
setitimer*	nan	1.0	1.0	1.0
setuid32*	nan	1.0	1.0	1.0
statfs64	nan	3.0	3.0	3.0

Figure 4.4: System Calls shared in Bashlite, Thetick and Normal in *NewNormalBenign50s*

Now, let's have a look at system calls that are shared across malware programs whereas the amount of them stays nearly constant (see Figure 4.4). One of these is *sendmmsg**

which appears in two of the four varying malware types (Bashlite and Thetick) and in three of the twelve attack datasets (for more details see column *NormalBenign50s* in Table 4.4). The sendmmsg system call is known to send multiple messages on a socket [31]. These system calls might stem from the Command & Control system structure that is shared between the Bashlite botnet and the Thetick backdoor. Figure 4.4 shows the appearance of some further system calls that occur in Bashlite and Thetick but are also part of the *Normal* dataset when looking at another 50 seconds system state of the Normal dataset (see *NewNormalBenign50s* in Table 4.4). These system calls might also be part of the standard radio frequency collection system as they occur in a similar frequency independent from the system state. When comparing the *NormalBenign50s* and *NewNormalBenign50s* in table 4.4 one can observe that there are further system calls that do only occur in the second system phase of the normal dataset (NewNormalBenign50s). Another system respectively context call that occurs in two malware datasets is setpgid respectively setpgid* which are responsible for setting the process group id of a process. The process can be added to an existing group id or a new group id can be created [35].

4.2 Shuffling consistency

Before passing the data to the models, the n-gram sequences respectively the one-hot-encodings are shuffled. Additionally, after each epoch of training and testing the data is shuffled again to keep the models from learning the order of the target labels. Following it is explained how it was checked that the data is consistent after shuffling. Although the time when the system calls are executed is not used for classifying attacks in the following, it is used to show that mixing the dataset and then splitting it into training and test sets doesn't lead to unbalanced data. The histograms of Figure 4.5 illustrate the randomness of the mixed data provided to the models by adding up the times of the first entries of each sequence in TF-(I)DF or one-hot-encodings whereas the left histogram shows the distribution over the time of the training set and the right histogram shows the number of datapoints occurring in different periods within the test data. Each histogram produces a total of five bars. This is due to the collection of the data which was performed, on one hand, while the system was used in its normal behaviour and on the other hand, while the system has been influenced by four different malware programs. The variety in the height of the bars can be explained by the number of attack phases per malware or benign state. On one hand, there is the leftmost and the bar second to last that stem from the benign state and the system state of the time when the system is infected by the RansomwarePoC malware. In both cases, only one phase was provided in the original dataset. On the other hand, the highest two bars represent the datapoints from the malware Bdvl and Bashlite that have been divided into four attack phases. Lastly, the rightmost bar stems from the data of the Thetick malware which has been split into two attack phases. The distribution of the bars of the two histograms is slightly different, but only to a small degree which can be explained by the randomness of the shuffling process.

Figure 4.7 shows the histograms when the shuffling process of the data is executed ten times on the TF-(I)DF. It shows that when repeating the process and taking the average results the differences between the training and testing sets get smaller. Further, the

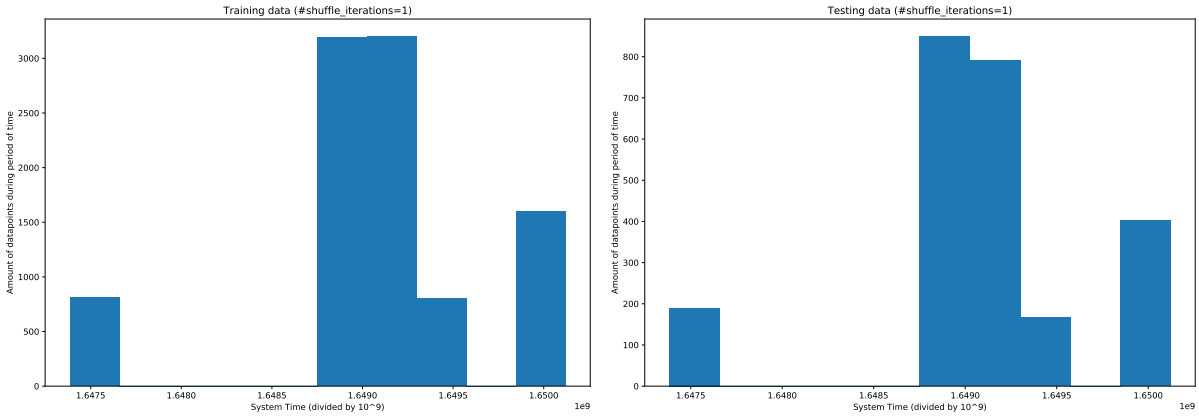

Figure 4.5: Figure shows two histograms with the data distributions in time after shuffling the dataset once (with tensor.randperm()) and splitting it into training and testing data

shuffling seems to work as the distributions of the training and test set to approach the distribution of the original dataset. Still, it is refrained from shuffling the input data ten times as the time is not a feature that is used for classifying an attack phase the argument goes that mixing the dataset once is enough such that the models won't learn wrong time dependencies. Another distribution that needs to be checked is the number of samples per attack phase and training set respectively testing set. Figure 4.6 shows the number of samples per attack phase and training respectively test set in an example of a randomly split dataset consisting of TF-(I)DF sequences. The red lines indicate the upper and the lower bounds of training set occurrences. It shows that the target classes are also balanced with smaller irregularities. The difference between the smallest and the biggest amount of data per attack phase of the training set in this example is at 2.27 percent. For the testing set, the difference of 8.95% is substantially higher but is ignored because in testing no model adjustment happens and therefore no overfitting is possible. The imbalancedness of the training set is neither treated as the difference is low and therefore in all likelihood, no model will overfit the targets which occur slightly more often in the dataset.

4.3 Preprocessing

The overall preprocessing is done by reading in the datasets of the different malware and balancing out the amount of data per dataset by taking the number of system call samples of the second smallest dataset (=19'645) and reducing the amount of data of the other datasets to this number except the smallest dataset (number of system calls=18'008). The main reason for this approach was to reduce the risk of overfitting which can occur when using imbalanced datasets [57]. Further, only one column (system call) of the datasets of a total of four columns (Time, Program, PID, & System Call) is considered. The columns Time, PID, and Program are ignored. This was done to build a more generalizable model which does not depend on a program name that can but mustn't be different from the observed malware programs in this dataset.

In total four different preprocessing techniques have been applied: N-grams, One-hot-encoding, TF-IDF, and positional encoding. In the following sub-sections, the implemen-

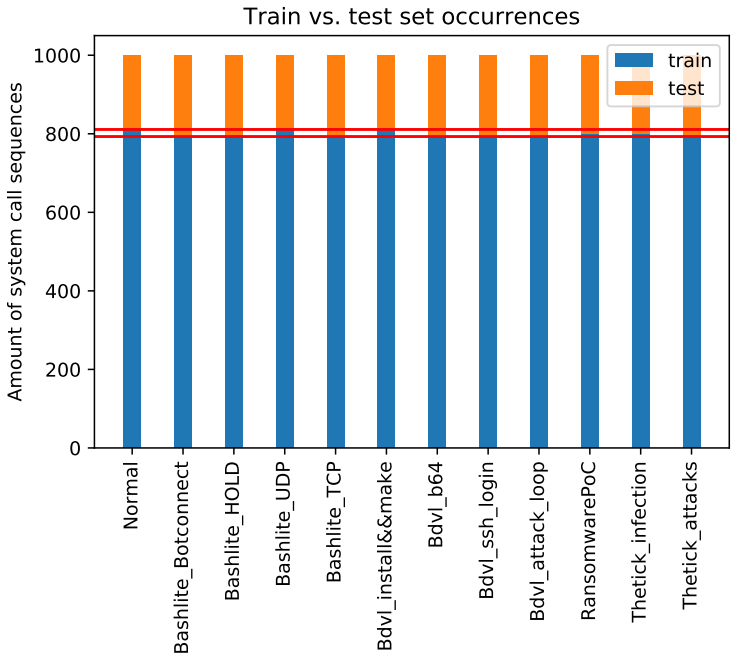

Figure 4.6: Distribution of system call sequences per attack phase after shuffling

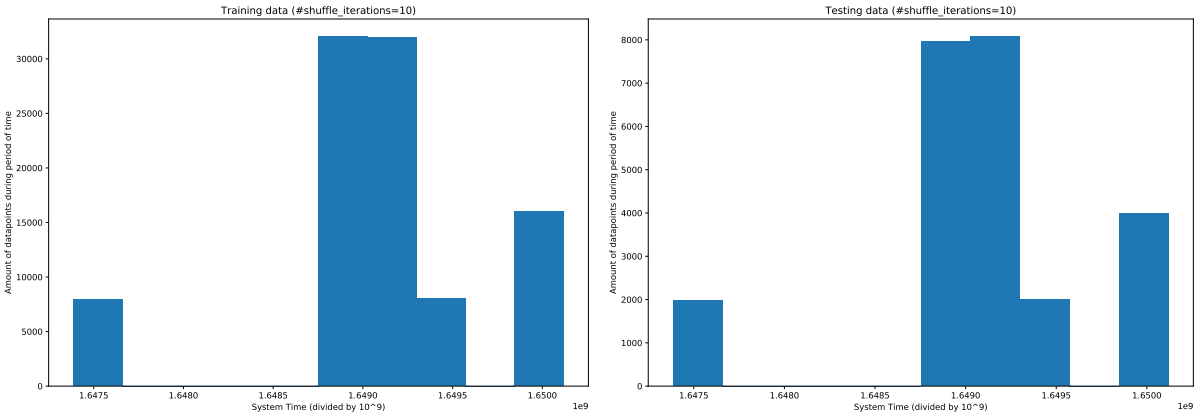

Figure 4.7: Figure shows two histograms with the data distributions sorted by time after shuffling the dataset ten times (with tensor.randperm()) and splitting it into training and testing data

Index	Original entry (system call)	3-gram
0	clock_gettime*	clock_gettime*:timerfd_settime:ioctl
1	timerfd_settime	timerfd_settime:ioctl:nanosleep*
2	ioctl	ioctl:nanosleep*:poll
3	nanosleep*	-
4	poll	-

Table 4.1: Example of turning a small dataset of system calls with length 5 into a dataset consisting of 3-grams

tation specifics of each of them are shortly described. After applying those techniques the different data samples need to be labeled according to the chosen learning goal; malware family or attack types classification. Finally, each of the datasets is randomly divided into a training and a test set. The training respectively the test set are split into parts consisting of 80% respectively 20% of the original data.

4.3.1 N-grams

System calls and n-grams are handled similarly as unique system calls can be interpreted as unigrams (1-grams). To prevent mixing up the sequence of n-grams a new dataframe is created where for each index instead of a system call the n-gram is saved. This is done by adding n-1 system calls to the original system call separated by a ":" (Example of a dataset of length 5 is shown in Table 4.1). When applying the n-gram preprocessing method the length of each sequence reduces by n-1 n-grams (can be seen in table 4.1). Thus, it might be needed to consider that when splitting a dataset into small sequences the number of system calls within a sequence could be reduced by half or more. E.g. when splitting the datasets into 1'000 sequences one sequence consists of 18 system calls whereas a transformation into 10-grams would render each sequence consisting of only 9 n-grams.

4.3.2 One-Hot-Encoding

For the one-hot-encoding of n-grams, a dictionary needs to be created. To avoid an extensive dictionary size and with unnecessarily long one-hot-encodings only system calls or n-grams existing in one of the malware or benign system state were considered in the dictionary. After collecting the present system calls, it is looped through the dictionary where for each term an array with the length of the dictionary size is created. These arrays are first filled with zeros. Then, with the help of the enumeration of the n-gram dictionary, a 1 is placed at the position at which index the system call has been saved in the dictionary.

4.3.3 TF-(I)DF

In addition to the TF-IDF which has been introduced in chapter 2.3.4 another formula was implemented whereas only the *idf* part of the equation has been changed. As the additional implementation doesn't make use of an inverse function it is hereafter called *TF-DF* respectively *df*:

$$df = log(NumberOfSequences - \frac{1}{NumberOfSequencesWithTermOccurrence})$$

When comparing the results of the two different functions within an input range of 1 to 1000 and the variable NumberOfSequences being fixed at 1000 one can observe two major differences in the curve shape (see Figure 4.8): On one hand, as indicated before the df(x) increases with higher numbers of x (=NumberOfSequencesWithTermOccurrence) while idf(x) sinks with bigger x. On the other hand, the df(x) curve is much steeper than the idf(x) curve.

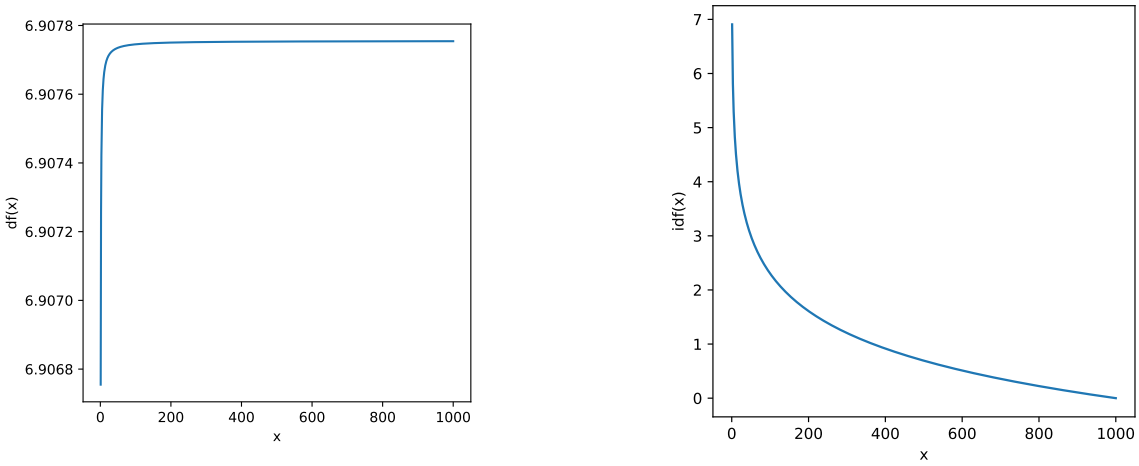

Figure 4.8: df(x) (left Figure) and idf(x) graph (right Figure)

Looking closer with the help of the log scale Figure 4.9 it shows that for the DF-function the biggest value differences arise between 10^0 to 10^1 whereas the graph nearly stagnates for x values between 10^2 to 10^3. Simultaneously, the curve in the IDF-function is linear when transforming the x-axis into the log scale for all x values.

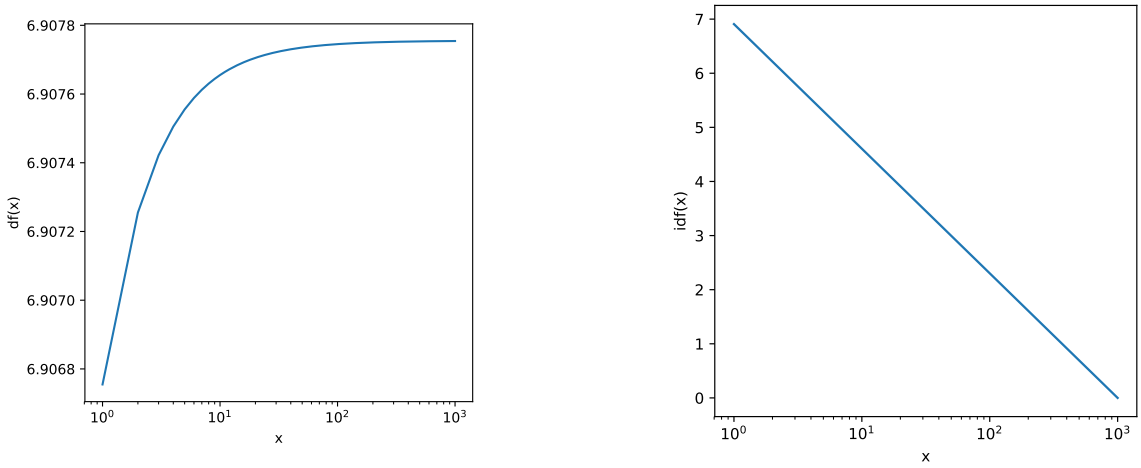

Figure 4.9: df(x) (left Figure) and idf(x) graph (right Figure) with log-scale on x-axes

As indicated at the beginning of the section the dataset needs to be split into sequences first. This makes it possible to calculate the importance of a term in a specific sequence with the help of the TF-(I)DF value. The sequences are called entities in the pseudo code (see **Algorithm 1**) where the calculation of the TF-DF is implemented for a dataset_dictionary where each dataset is the data of one specific system state (see targets_dict) consisting of benign or malware data:

4.4 Models' specifics

The models used in this work have already been briefly explained in 2.1. The models have been implemented with the help of the PyTorch Library. Before going into the implementation details of each model the similarities in the implementations of the models are shortly discussed here. All implemented models make use of the same optimizer and the same amount of input and output neurons. The chosen optimizer is the Stochastic Gradient Descent respectively Batched Stochastic Gradient Descent optimizer for the models that require batch processing. This results in two ways of training the models: While the Neural and the Autoencoder network has been run for 10'000 epochs the LSTM, CNN, and Transformer models are run for 100 epochs. System calls have been split into several sequences which are in the following called *tf_idf_seq_len*. The batch_size is chosen equivalently to the sequence length of given inputs if the TF-(I)DF preprocessing method is performed. Else, in the one-hot-encoding preprocessing sequence lengths respectively batch sizes between 25 and 150 are selected whereas models that have made use of TF-DF preprocessed inputs have been trained with sequence lengths respectively batch sizes ranging between 10 and 40. Additionally, the sequence length is utilized to decide how many system call steps between two sequences are ignored (step_size=round(seq_len/10)). Therefore, when having twenty system calls and the step_size would be chosen to be two which would result in ten sequences. This was done to diminish training time by hindering the processing of all data. The number of training steps can therefore be computed by

Algorithm 1 TF-DF value collection example

```
1: inputs = []
2: targets = []
3: targets_dict = {"Normal":0,"Bashlite":1,"Bdvl":2,"RansomwarePoC":3,"Thetick":4}
4: for dataset_name in dataset_dictionary do
5:     nr_of_entities_term_appears = {"ioctl": 0, "poll": 0, ..., "nanosleep": 0}
6:     term_frequencies_dict = {"ioctl": [], "poll": [], ..., "nanosleep": []}    ▷ at the end of this loop each array will have the length of the amount of entities
7:     for all entities do
8:         Count occurrences of the term in entity
9:         for all terms t over all entities do
10:            if if term t in current entity then
11:                tf = term_frequency_of_entity/overall_term_frequency
12:                Append tf to the list of term t in term_frequencies_dict
13:                Increase count of the current t entry of nr_of_entities_term_appears
14:            else
15:                Append 0 (=tf) to the list of term t term_frequencies_dict
16:            end if
17:        end for
18:    end for
19:    for i in range(0, amount_of_entities) do
20:        tf_idf_scores_list = []
21:        for all terms t that exist in the datasets do
22:            tf = term_frequencies_dict [t][i] ▷ gets the correct term frequency of term t and current entity
23:            if nr_of_entities_term_appears[term] != 0 then
24:                idf = Log((amount_of_entities) -1 /nr_of_entities_term_appears[t])
25:            else
26:                Set idf to zero
27:            end if
28:        end for
29:        tf_idf_val = tf*idf
30:        Append tf_idf_val to tf_idf_scores_list
31:    end for
32:    Append tf_idf_scores_list to inputs
33:    Choose target of targets_dict with the help of dataset_name and append its numeric representation to the targets list
34: end for
```

the following formula:

$$training\_steps = round(\frac{amount\_of\_datapoints * 0.8 \cdot number\_of\_targets \cdot nr\_epochs}{tf\_idf\_seq\_len \cdot step\_size \cdot batch\_size})$$

In that way, the training of the network is repeated at minimum around 500 times and at most up to 4'000 times in the TF-IDF case respectively when using one-hot-encoding the training repetition lies between around 760 and 6'900 times. Therefore, the number of training steps of the regularly processed models is at a higher number than the number of training steps of batch-processed models. Another difference between the two types of models is implemented in the TF-(I)DF preprocessing technique. As the batch processing models are sequence models the TF-IDF arrays have been preprocessed such that a sequence of TF-(I)DF arrays are assigned to a target class and given to each batch training step together with TF-(I)DF arrays of other classes. Whereas the standard processing takes in the TF-(I)DF arrays of each sample per epoch there is only one TF-(I)DF array that is handed to the model per target class.

Further, the Stochastic Gradient Descent brings two caveats: On one hand, the order of the samples matters especially when training with batches which requires shuffling the data after each epoch, on the other hand, the gradient directions may alternate. This caveat can be partly eliminated by adding to the matrix of the standard weight update a momentum learning variable which is multiplied by the difference of the weights of the previous steps. In this way, the direction of the last weight update can be remembered and is then accumulated. The best-fitting momentum value is in some of the models found by hyperparameter-tuning. Another mutual hyperparameter variable is the learning rate. Generally, a learning rate defines the speed of convergence of the gradient descent method. However, if the learning rate is set too high the minimum of the loss function might not be reached due to divergence.

Whereas the number of input neurons is always the size of the dictionary set over all datasets the number of output neurons depends on the loss method and the number of targets. Two loss methods have been implemented: Binary Cross Entropy (BCE) Loss was used for Binary Classification and Categorical Cross Entropy (CCE) Loss for Categorical Classification. The BCE loss produces one value between -1 and 1 which can be interpreted similarly to a probability value. If the value is lower than 0 class A was classified and if it is higher or equal class B was classified. As this approach only produces one output also one output neuron is used for all models. In the case of the CCE loss, a probability for each possible output is computed so the output neurons must be reduced to the number of targets. In the end, only CCE has been evaluated as it also has yielded promising results for binary classification after reducing the results to a binary classification problem.

In the following sub-sections, the models that were used in this thesis are revisited and implementation details are discussed. Further, it is indicated which hyper-parameters were tested. Table gives an overview over the tested hyper-parameters.

Models	NN	Autoencoder	CNN	LSTM	Transformers
Learning rate	0.1,0.05,0.01,0.001,0.0001			0.1,0.01,0.001	
Number of sequences (for TF-(I)DF)	400,550,700,850 and 1000		850	400	850
Number of embeddings	N.A.				7
Hidden neurons	10,30,50		700		50, 70 & 700
Hidden layers	N.A.			3, 4 & 5	N.A.
Dropout	N.A.				0.1
MultiAttentionHeads	N.A.				5
Sequence length / Batch size	N.A.		10,20,30,40 (TF-(I)DF) resp. 25,50,100,150 (1-Hot)		
Encoder Dimension	N.A.	1,3,5,7	N.A.		
Momentum	0.5,0.8,0.9,0.99,1		0.99		
Epochs	10000		100		

Table 4.2: Overview of datasets and attack phases used for categorical classification tasks

4.4.1 Neural Network

Apart from the given input and output neurons, the number of hidden neurons of the neural network was chosen through hyperparameter-tuning. Sometimes, the ultimately selected amount of hidden neurons also depends on the preprocessing technique used and other input parameters such as tf_idf_seq_len.

4.4.2 Convolutional Neural Network

The implementation of the CNN is done depending on the given preprocessing technique. Average-Pooling was used independently from the preprocessing technique, although in related work there have been seen different approaches. However, the preprocessing with TF-(I)DF conducted in this work differentiates from the one used by the authors of [54] as in comparison to this thesis where it was chosen to stack the TF-IDF arrays to 2-dimensional matrices whereas in the latter work they weren't stacked and therefore given to the model as 1-dimensional arrays. Further, two pooling layers with dimension 2x2 were chosen that halve the given input dimensions whereas the convolutional layers each have a kernel size of (3,3) and padding and stride equal to 1. Given these convolutional layers, the input dimension equals the output dimensions. Therefore, considering only the pooling layers the last linear layer must be of the following dimension (lin_dim, O) (see following equation) where lin_dim is computed with the Q1 and Q2 input dimensions and height dimension O is equal to the output dimension:

$$lin\_dim = \lfloor (Q1/4) \rfloor \cdot \lfloor (Q2/4) \rfloor$$

4.4.3 Autoencoder

The Autoencoder was implemented with a total of six layers of which the encoder comprises four, the decoder consists of a single layer and lastly, one layer is shared by the encoder and the decoder. The different numbers of neurons of the in-between layers of the encoder are selected to be in accordance with the input dimension (I). Going into deeper

layers of the encoder the number of neurons decreases. The first inbetween layer has *round(I/2)* input neurons and *round(I/4)* output neurons. The last layer of the encoder has *round(I/8)* input neurons and its output parameter is the encoder output dimension shortened *E_D*. This dimension is chosen through hyperparameter-tuning. Finally, the decoder takes those E_D output neurons and linearly transforms the pre-defined output dimension.

4.4.4 LSTM

The only LSTM-specific parameter is the number of hidden layers the model shall consist of. In this thesis 3, 4, and 5 have been evaluated as amount of hidden layers.

4.4.5 Transformer

Whereas the LSTM model requires sequence lengths to function the Transformer (respectively the class TransformerEncoderLayer torch) demands the embedding dimension to be divisible by the number of heads chosen for the MultiheadAttention. As the embedding dimension is the input dimension of the Transformer model it is again chosen to be the length of the dictionary of the input. With different n in the n-grams this length changes. Therefore, in the original implementation, it is made sure that the input dimension is changed according to the number of heads such that the requirement is fulfilled. Unfortunately, experiments, where the n of n-grams is higher than 1, have shown that the memory resources of the training infrastructure have been too little to train sequence models with 2-grams or higher. As has been stated, the model needs to be adaptable to enable the hyper-parameter tuning of the number of heads. This is achieved by taking the modulo from the following expression and then subtracting the remainder from the dictionary length. The resulting value is set to be the input dimension.

$$remainder = dictionary\_length \mod num\_heads$$

The author of [46] proposes that more heads are generally better, but that they can get redundant, therefore it is refrained from testing a different number of heads in hyper-parameter tuning. Additional hyper-parameters are the embeddings' length and the dropout probability of the Transformer. However, due to time constraints, it is also forgone to evaluate those parameters. Equivalently to the number of encoder and decoder layers that were chosen by the authors of [62] in this thesis the amount of encoder layers has been chosen equally whereas the decoder layer has been implemented as a linear layer for simplicity.

4.5 Classification Approaches

Implementing categorical malware classification was executed in two ways: First, only the initial attacks of malware have been used to classify if malware is installed on an IoT

Attack Phase	Original Dataset	Attack Dataset Name	Initial Attack
No attack	Normal	Normal	-
Connection of Bot to Botnet	Bashlite	Bashlite_Botconnect	yes
HOLD flood	Bashlite	Bashlite_HOLD	no
UDP flood	Bashlite	Bashlite_UDP	no
TCP flood	Bashlite	Bashlite_TCP	no
install and make	Bdvl	Bdvl_install&&make	yes
build super.b64: Installation of Backdoor	Bdvl	Bdvl_b64	no
SSH login	Bdvl	Bdvl_ssh_login	no
Stealing files and un-/hiding directories	Bdvl	Bdvl_attack_loop	no
Encryption of directories	RansomwarePoC	RansomwarePoC	yes
Installation of backdoor	Thetick	Thetick_infection	yes
File Transfers	Thetick	Thetick_attacks	no

Table 4.3: Overview over datasets and attack phases used for categorical classification tasks

device or not (see Greedy Categorical Classification). Second, more attack phases were introduced to the classification process (see sub-section Bottom-Up Categorical Classification). Table 4.3 gives an overview of what happened in the different attack phases and which phases were considered initial attacks. Latter attacks were used to find out what malware is installed on the system (see Greedy Malware Classification).

4.5.1 Greedy Malware Classification based on initial attacks

The initial approach for malware classification was to only consider malware data while it was installed on the target's device. The intuition behind the *greedy malware classification* is that one could use it in real-time to encounter the first attack of a system. This approach is referred to as greedy as it only cares if the malware was installed on a system or not and doesn't consider in which installation/attack phase the program is. Reasoning that it could be of interest in which attack phase a program is a further approach has established. This approach is explained in the next section.

4.5.2 Attack Classification based on different attack phases

The attack classification based on different attack phases takes into consideration which phase of attack malware respectively the attacker is executing. Based on the chosen dataset this results in twelve different attack phases including a bening case. As indicated in "Data Exploration" the original dataset consists of different stages of attacks within the malware datasets. Figure 4.3 shows an overview of the attacks of the different malware. The attack phase classification is used mainly in this thesis. However, starting the work the main focus was lying on malware detection. Therefore, the results of the attack classification are in this work also used to show the performance of a malware detection classification of the underlying model. This is reported reduced to show to some extent how the models could perform in malware detection however a model might learn malware specifics better if the model would only need to learn the malware behavior.

System Call/Dataset	NormalBenign10s	NormalBenign50s	NewNormalBenign50s
getdents*	Normal	Normal	Normal
getpgrp	RansomwarePoC	RansomwarePoC	RansomwarePoC
lseek	Normal	Normal	Normal
lseek*	Normal	Normal	Normal
open	Normal	Normal	Normal
open*	Normal	Normal	Normal
setgid32	Bdvl_ssh_login	Bdvl_ssh_login	Bdvl_ssh_login
symlink*	Bdvl_install&&make	Bdvl_install&&make	Bdvl_install&&make
symlink	Bdvl_install&&make	Bdvl_install&&make	Bdvl_install&&make
unlinkat	RansomwarePoC	RansomwarePoC	RansomwarePoC
exit	Bashlite_TCP	Bashlite_TCP & Normal	Bashlite_TCP & Normal
setpgid	RansomwarePoC	RansomwarePoC& Thetick_attacks	RansomwarePoC& Thetick_attacks
setpgid*	RansomwarePoC	RansomwarePoC& Thetick_attacks	RansomwarePoC& Thetick_attacks
gettid	Bdvl_attack_loop	Bdvl_attack_loop& Bashlite_UDP & Thetick_attacks	Bdvl_attack_loop& Bashlite_UDP & Thetick_attacks& Normal
tgkill*	Bdvl_attack_loop	Bdvl_attack_loop& Bashlite_UDP & Thetick_attacks	Bdvl_attack_loop& Bashlite_UDP & Thetick_attacks& Normal
perf_event_open*	Bdvl_attack_loop	Bdvl_attack_loop& Bashlite_UDP & Bashlite_HOLD& Thetick_attacks	Bdvl_attack_loop& Bashlite_UDP & Bashlite_HOLD& Thetick_attacks& Normal
perf_event_open	Bdvl_attack_loop	Bdvl_attack_loop& Bashlite_UDP & Bashlite_HOLD& Thetick_attacks	Bdvl_attack_loop& Bashlite_UDP & Bashlite_HOLD& Thetick_attacks& Normal
statfs64*	Bashlite_TCP	Bashlite_TCP & Bashlite_UDP & Bashlite_HOLD& Thetick_attacks& Bdvl_b64	Bashlite_TCP & Bashlite_UDP & Bashlite_HOLD& Thetick_attacks& Bdvl_b64& Normal
kill*	Bdvl_b64	Bdvl_b64 & Bashlite_UDP& Bashlite_HOLD& Thetick_attacks& Normal	Bdvl_b64 & Bashlite_UDP& Bashlite_HOLD& Thetick_attacks& Normal
getpeername*	Bdvl_attack_loop	Bdvl_attack_loop & Bashlite_UDP & Bashlite_HOLD& Thetick_attacks	Bdvl_attack_loop & Bashlite_UDP & Bashlite_HOLD& Thetick_attacks& Normal

4.5.3 Adversarial attack

For evaluating an adversarial attack the following code was implemented that modifies the test data *X_val* such that system calls that only occur in the benign system state are added to the malware samples according to its original distribution (see **Algorithm 2**). A similar approach was implemented by the authors of [4] where only a fixed fraction (1-3%) of unique system calls was added to the test data. In the following pseudo-code of the implementation of this work, X_val is a tensor of shape [amount_of_samples, amount_of_system_calls].

Algorithm 2 Unique normal Syscalls Attack

```
benign_syscalls_dict = {"syscall1":(tf_idf_array_index, avg_frequency), ...}
for index, tf_idf_array in enumerate(X_val) do
    if if target, not a benign system state then
        for syscall, tuple_value in benign_syscalls_dict do
            tf_idf_array_index, avg_frequency = tuple_value
            probabilities_list = []
            Creating probability distribution with avg_frequency
            counter = 0
            amount_of_occurrences_list = []
            while counter < amount_of_entities do
                Choosing amount_of_occurrences based on probability distribution
                Append amount_of_occurrences to amount_of_occurrences_list
                counter += 1
            end while
            Computing tf_idf_value based on amount_of_occurrences_list
            X_val[index,tf_idf_array_index] += tf_idf_value
        end for
    end if
end for
```

Chapter 5

Evaluation

For the evaluation of the models, a train-test-split of 80/20 was performed on the data. Five models were trained and tested with hyperparameter tuning to find the best parameter configuration of the models. The number of hyperparameters tested depended on the training time of the models whereas some types of hyperparameters are unique to one model e.g. the number of attention heads in the Transformer model.

5.1 Results depending on models and preprocessing techniques

Table 5.1 shows an overview of the results of evaluating the models in combination with different preprocessing techniques when performing hyper-parameter tuning in attack phase classification. Also, it shows the best hyper- and preprocessing parameters such as the optimal *Number of sequences* that are generated from the input data before its sequence values are transformed into TF-(I)DF arrays. The first observation that can be done is that the simple Neural Network model with the TF-DF preprocessing technique significantly outperforms every other model that was evaluated independently from their preprocessing techniques and the sequence models (CNN, LSTM & Transformer). When looking closer, Table 5.1 shows that the sequence models classify all attacks as being from one single attack phase which results in a very low F1-score. Older experiments have shown, that LSTM can yield an accuracy of 52.88% if instead of the attack phase classification task malware detection is performed where only five classes are considered and the input data is narrowed down to the initial attack phase of each malware dataset (greedy method). Whereas Neural Network and Autoencoder weren't evaluated with one-hot-encoded inputs method the best model that is served one-hot-encoded features is the CNN, although, only yielding an F1-score of 7.75% which is even slightly worse than randomly choosing a class. Although the Autoencoder (AE) seems to have a high score it is unfortunately unstable. After repeating the attack phase classification task with the best hyper-parameter configuration it F1-score decreased to 27.61%. Still, the Autoencoder and especially the Neural Network with the TF-(I)DF preprocessing techniques perform significantly better than the sequence models which sometimes even perform worse than choosing the attack

	NN						AE	CNN		LSTM		Transformer	
Preprocessing technique	TF-IDF			TF-DF			TF-DF	TF-DF	1-Hot	TF-DF	1-Hot	TF-DF	1-Hot
N-gram	1-gram	2-gram	3-gram	1-gram	2-gram	3-gram	1-gram	1-gram					
Momentum	1	0.9	1	0.8	0.99	0.8	0.5	0.99					
Number of sequences (TF-(I)DF)	700	550	700	700	550	1000	850	850	N.A.	400	N.A.	850	N.A.
Learning rate	0.001	0.1	0.001	0.05	0.1	0.1	0.001	0.001	0.01	0.1	0.1	0.01	0.01
Sequence Length / Batch Size	N.A.							20	100	20	50	30	25
Hidden dimension	30	10	50	50	10	10	50	700	700	700	700	70	50
F1-score (in %)	61.0	61.28	64.13	100			93.64	7.31	7.75	2.17	1.32	4.11	3.68

Table 5.1: Results overview with the best hyper-configurations

phase randomly. The fact that the Term Frequency-Document Frequency (TF-DF) significantly outperforms the TF-IDF is surprising but can be explained when looking at the distribution of the frequencies of the different system calls. The differences between these methods are introduced in the chapter *Implementation Details* and the argumentation for the performance differences is stated in the next chapter **Discussion**.

The performance stability of a model is evaluated with the example of the Neural Network model combined with the Term Frequency-Document Frequency (TF-DF) and 1-gram data preparation method by repeating the classification in five independent iterations. Whereas this experiment yielded an average F1-score of 99.23% all five classifications resulted in an F1-score of at least 99.13% and at most 99.62%. Now let's consider another type of stability, by looking for a pattern of misclassifications occurring over multiple iterations. Figure 5.1 shows that in ten cases of a total of 18 different confusion errors a misclassified sequence only occurred in one of the five iterations. Apart from those cases where the confusion value is 1 a confusion between Bashlite_HOLD (predicted class) and the RansomwarePoC (true class) attack occurred, which is marked with the value 3, is the only example wherein a single iteration of more than one sequence was misclassified but hasn't been confused in any other iteration. Simultaneously, seven misclassifications happened in at least two of the five iterations. Therefore, one needs to partly reject the stated stability hypothesis. Therefore, now it must be observed closely to see if at least a tendency of the model can be reasoned. Beginning with a closer look, it can be seen that the benign system state was never confused with an attack phase which leaves the accuracy at 100% if the result would be reduced to binary anomaly detection. Further, it can be observed that except between the two attack phases of Thetick the confusion appears to happen across malware types only which can be explained by the reasoning that there are more similarities within two system states if in both states e.g. two different malware are installed in comparison to the case where in one state an installation and in another and an attack of only one specific malware is carried out.

By reducing the beforementioned confusion matrix from an attack phase classification to a malware detection problem which is shown in Figure 5.2 is produced where can be seen that, apart from two confused samples between RansomwarePoC and Bdvl, there only happened confusions between Bashlite and the other malware but not in between the other malwares. The confusion with Bashlite and the other malwares and the confusion between Bdvl and RansomwarePoC could arise because every malware has been executed through another device and therefore each malware can be seen as having a command and control structure. Therefore, it suggests that the model would randomly guess the malware when an attack through the command and control structure needs to be classified. Continuing, the argument that the model makes decisions that can be explained seems to hold true

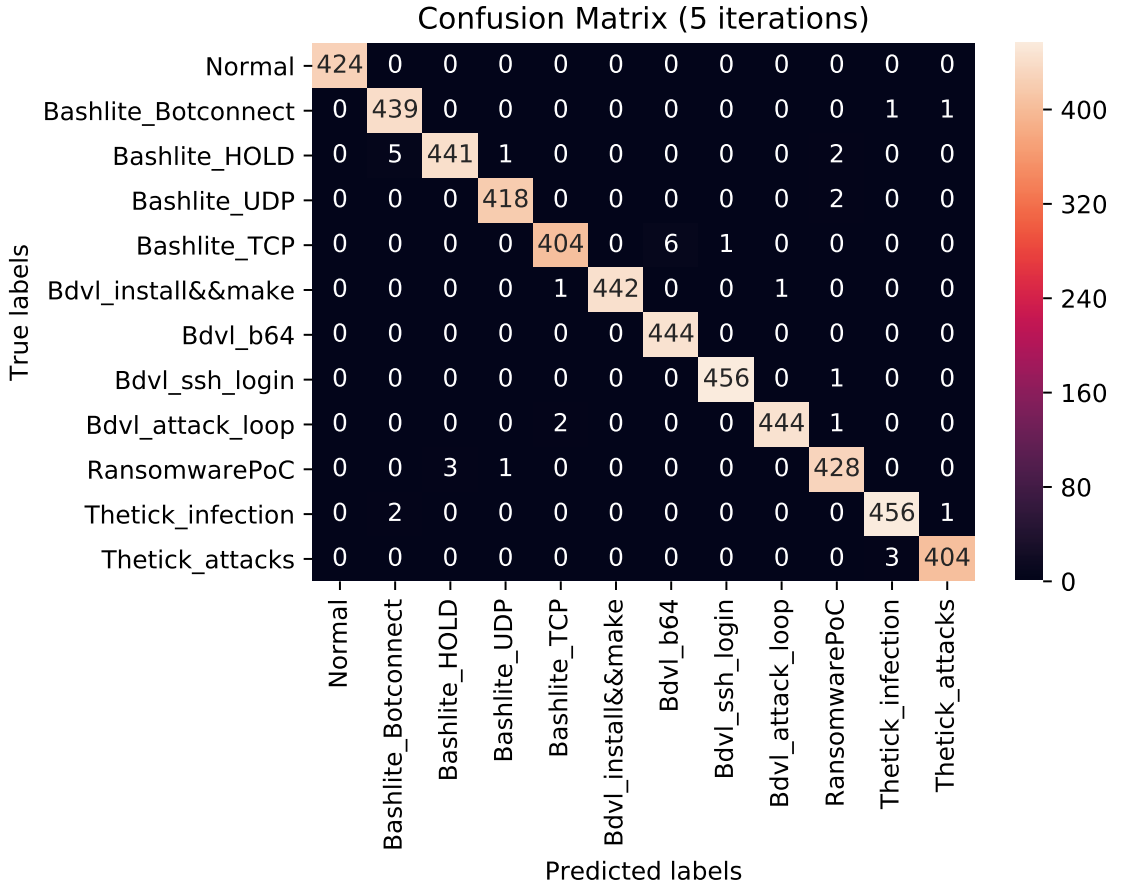

Figure 5.1: Added up Confusion matrix of five iterations of attack phase classification with the help of the best-performing Neural Network configuration with the preprocessing techniques TF-DF and 1-gram

according to the beforehand reasoning when neglecting the small confusion between the two Thetick attack phases whereas the confusion between Bdvl and RansomwarePoC could also be explained by the shared command and control structure.

While the confusion matrix of five iterations showed the stability of the model in this paragraph the results of one such iteration are discussed. With the help of Table 5.2 the confusion matrix of the NN-model combined with 1-gram and TF-DF preprocessing is shown but also the F1-scores per each class are illustrated. Further, the F1-scores of the deducted malware classification (see column *F1-score over 5 classes*) and the anomaly detection (see column *F1-score over 2 classes*) are reported per class. As has been expected from the above 5-iteration model performance the score that classifies if the systems state is benign or not reaches again a F1-score of 100% whereas the macro F1-score for malware detection is 99.4% which has been calculated by taking the mean from the values captured in the *F1-score over 5 classes* column. Although, the amount of samples per malware hasn't been balanced the weighted average F1-score matches with the macro F1-score both are at 99.4%. Figure 5.5 shows the reduced confusion matrix which illustrates the distribution of confused samples for malware detection. Whereas it also shows that the system call sequences stemming from the Thetick malware's system state were never

confused with any other malware nor the benign system state. Overall, the attack phase classification yielded an average macro and a weighted average F1-score of 99.2 percent.

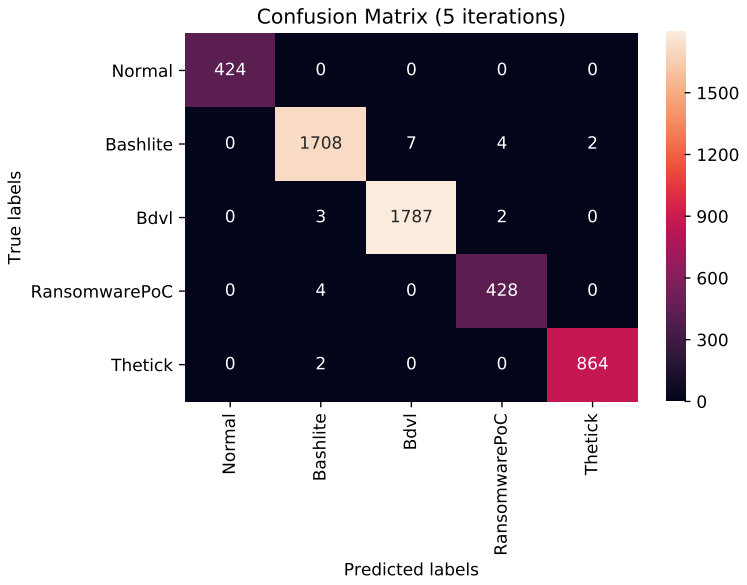

Figure 5.2: Added up Confusion matrix of five malware classification iterations with the help of the best-performing Neural Network configuration with the preprocessing techniques TF-DF and 1-gram

Comparing the 1-gram performance of the best-configured parameters of the Neural Network model combined with the TF-IDF preprocessing technique shows a significantly worse classification performance with an F1-score of 53.33%. Although, the F1-score is worse the True Negative Rate is 96.66% which means there is a very low amount of false positives in comparison to the fraction of true negatives. Therefore, e.g. an attack sequence is much more often classified as not belonging to attack type A whereas the actual class is not A than it is falsely classified as being of attack type A whereas the actual class is different from class A. Still, looking at the confusion matrix (see Figure 5.3) we see that the model is not able to differentiate between benign and infected states as often the normal class is predicted although the class under evaluation belongs to an attack phase of mostly Bdvl malware but also other malware types. Consequently, this attack phase classification can't be reduced to an anomaly detection classification. Further, it is analyzed if the TF-IDF preprocessing technique could be a technique that is usable for malware detection. Therefore, we have a look at the malware confusion matrix (also in Figure 5.3) that again has been deducted from the attack phase classification confusion matrix. It shows that apart of the confusion of the normal phase with other malware the confusion is not very high between the malware themselves. Still, this result needs to be put in perspective as the normal behaviour was often confused which could have prevented confusion between the malware types. Further, the apparent misclassification towards the normal behaviour might not be constant as the model might overclassify an attack phase instead of the benign phase in another run. An indication of that is presented in Figure 5.4 where the TF-IDF preprocessing approach was combined with 3-grams. It can be observed that instead of the seen multiple misclassifications of the normal system state the model often classifies a system call sequence as being part of the ssh login which is

	Predicted class												F1-score over 12 classes	F1-score over 5 classes	F1-score over 2 classes
	Normal	Bashlite				Bdvl				RansomwarePoC	Thetick				
	-	Botconnect	HOLD	UDP	TCP	install&&make	b64	ssh login	attack loop	-	infection	attacks			
Normal	89	0	0	0	0	0	0	0	0	0	0	0	1.000	1.000	1.000
Bashlite_Botconnect	0	74	0	0	0	0	0	0	0	0	0	0	0.993	0.993	1.000
Bashlite_HOLD	0	1	77	0	0	0	0	0	0	1	0	0	0.987		
Bashlite_UDP	0	0	0	104	0	0	0	0	0	1	0	0	0.995		
Bashlite_TCP	0	0	0	0	79	0	1	1	0	0	0	0	0.981		
Bdvl_install&&make	0	0	0	0	1	93	0	0	0	0	0	0	0.995	0.994	
Bdvl_b64	0	0	0	0	0	0	93	0	0	0	0	0	0.995		
Bdvl_ssh login	0	0	0	0	0	0	0	70	0	1	0	0	0.986		
Bdvl_attack loop	0	0	0	0	0	0	0	0	92	0	0	0	1.000		
RansomwarePoC	0	0	0	0	0	0	0	0	0	85	0	0	0.983	0.983	
Thetick_infection	0	0	0	0	0	0	0	0	0	0	95	1	0.995	1.000	
Thetick_attacks	0	0	0	0	0	0	0	0	0	0	0	88	0.994		

Table 5.2: Confusion matrix of Neural Network 1-gram & TF-DF attack phase classification with eleven benign states

performed while the Bdvl malware is active. Still, the 3-gram approach performs slightly better than the 1-gram approach when looking at the attack phase classification where in the 3-gram approach an F1-score of 55.58% is reached.

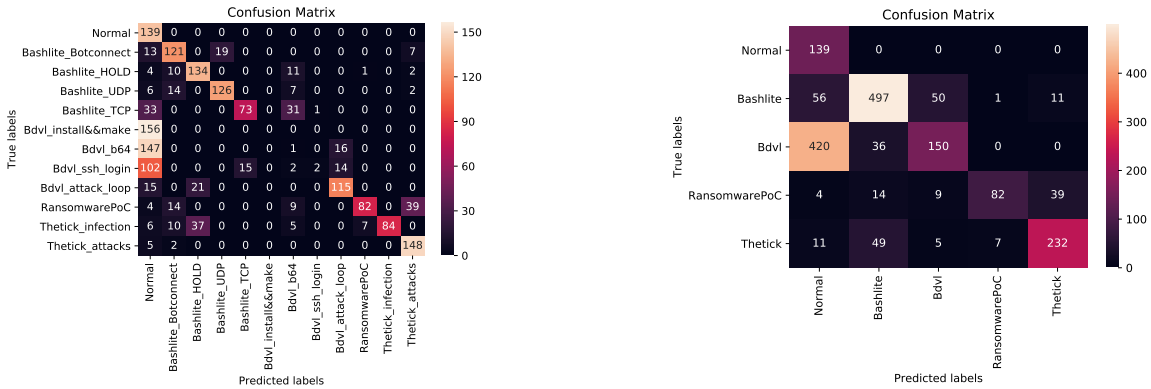

Figure 5.3: TF-IDF confusion matrices with NN model and 1-gram: Attack phase classification (left) and Malware classification (right)

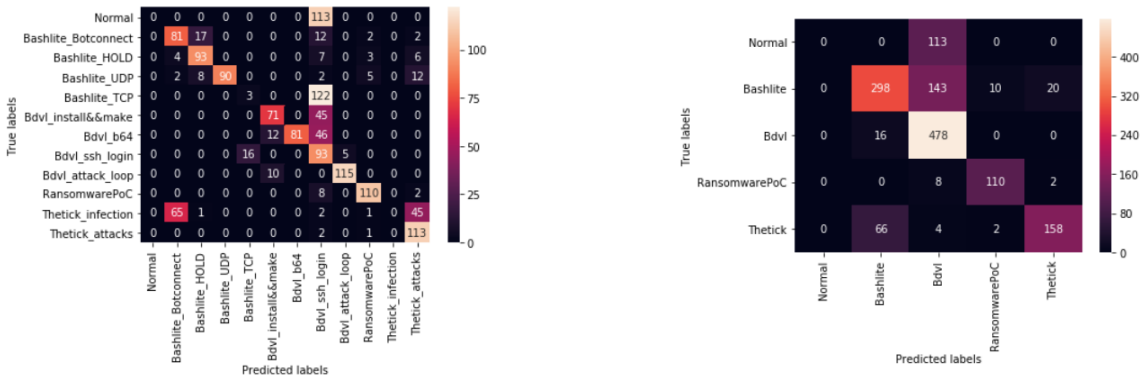

Figure 5.4: TF-IDF confusion matrices with NN model and 3-grams: Attack phase classification (left) and Malware classification (right)

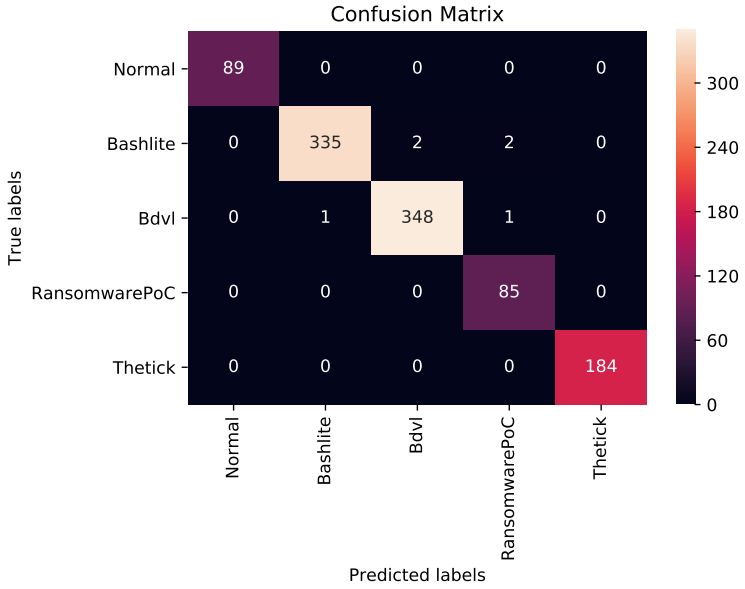

Figure 5.5: Confusion matrix of malware classification deducted from the attack phase classification confusion matrix from Table 5.2

5.2 Results of different datasets

As could be seen in the *Implementation Details* chapter the occurrence of system calls depends partly on how much data the model is provided with. In this subsection, we want to find out if the models' classification performance changes when different and/or more data is provided. To achieve this the Neural Network is trained with data from four different datasets:

- Dataset 1: Each attack phase is recorded for 10 seconds and a balanced amount of data is given to the model.
- Dataset 2: Also each attack phase is recorded for 10 seconds, but the benign state dataset is changed to another period. Also balanced amount of data.
- Dataset 3: Half of the attack phases are recorded for 10 seconds and a half is recorded for 50 seconds. Therefore, the model receives an unbalanced amount of data.
- Dataset 4: The same distribution of unbalancedness as in Dataset 3 but the benign data is changed to another time window.

The above datasets are first preprocessed into unigrams and then transformed into sequential vectors with the help of the TF-IDF technique. Table 5.3 shows the best configurations and classification results for the four different datasets. The two values separated by a slash in the number of sequences row stem from the fact that the length of the sequences is not changed throughout a dataset and therefore attacks recorded for fifty seconds have more sequences (value on the right side of the slash) than when the attack phase was collected for ten seconds (value on the left side of the slash). Datasets 1 & 2 only have

	Dataset 1	Dataset 2	Dataset 3	Dataset 4
Date of benign state recording	16.03.2022	07.04.2022	16.03.2022	07.04.2022
Time of system recording	10 seconds	10 seconds	10/50 seconds	10/50 seconds
Learning rate	0.001	0.001	0.001	0.001
Number of sequences (10s/50s)	755	595	935/5'429	436/2'533
Sequence Length	26	33	21	45
Hidden Dimension	30	50	90	50
Momentum	1	1	1	0.99
F1-score	61.0%	56.4%	61.46%	59.11%

Table 5.3: Evaluation of different datasets and their classification results with TF-IDF and 1-gram preprocessing

one amount of sequences as the maximum period of its attacks stay 10 seconds throughout all attacks. All in all, the table shows that the classification results measured with the F1-score stay relatively constant despite changing datasets. Especially, it stands out that changing the benign dataset seems to have a bigger influence than increasing the dataset of half of the attack phases. However, as the classification only has been performed once per the best hyper-parameter configuration the absolute performance values have to be put into perspective. Throughout the thesis *Dataset 1* has been used for evaluating the different models.

5.3 Results of benign phase classification

To make sure that including the context system calls doesn't result in a lack of generalization it was tested how well the model performs when being served with twelve different datasets of the benign system state instead of eleven different attack phase data one benign phase data. The result of classifying benign phases with NN and TF-DF gives an F1-score of 94.09% which is only slightly worse compared to the result of the beforehand discussed attack phase classification (99.2%). This might indicate that the model also to some degree learns the time-depending system state. The confusion matrix 5.6 illustrates the benign system states that the system couldn't differentiate. The misclassifications have probably occurred due to a repeating system's behaviour. However, the chosen benign sequences (*Normal0* to *Normal11*) all stem from a different hour and/or day period.

Evaluating, if the confused datasets are similar is done by looking at the system call occurrences of different system calls. The Figures 5.7 show the distribution of system call occurrences in an ordered manner where the lowest occurring system call is the first value on the x-axis. The left Sub-Figure illustrates these occurrences of all system calls found in the four benign phases. There it seems that the confused benign phases *Normal3* and *Normal6* are not very similar although they have been confused the most. The right Sub-Figure within Figure 5.7 again shows the same distribution but focuses on the 61 lowest occurring system calls. Further, it can be noticed that the two confused phases are farther apart in this region than compared with the right Sub-Figure, where the distribution of every system call has been plotted. Therefore, one could infer that the similarity between

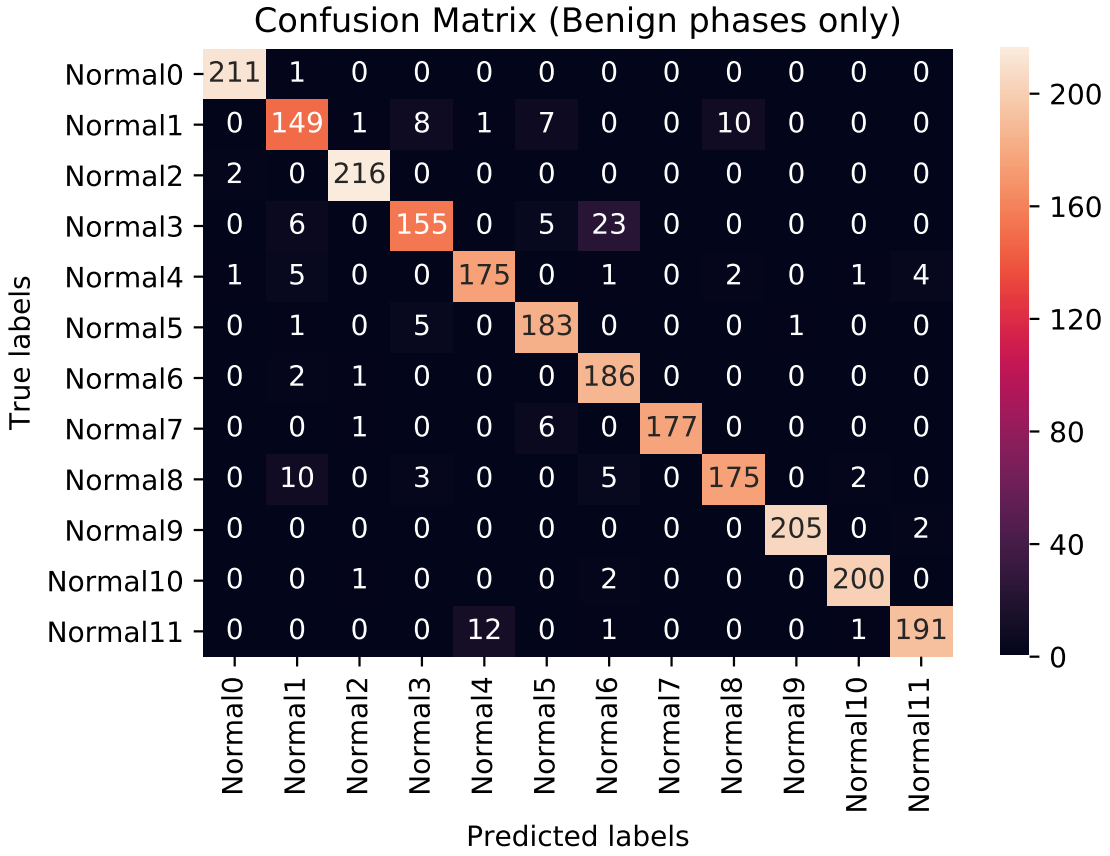

Figure 5.6: Added up Confusion matrix of five iterations of attack phase classification with the help of the best-performing Neural Network configuration with the preprocessing techniques TF-DF and 1-gram

the distribution of low-occurring system calls determines the classification of a NN with the TF-DF preprocessing technique. This will be further discussed in the next chapter.

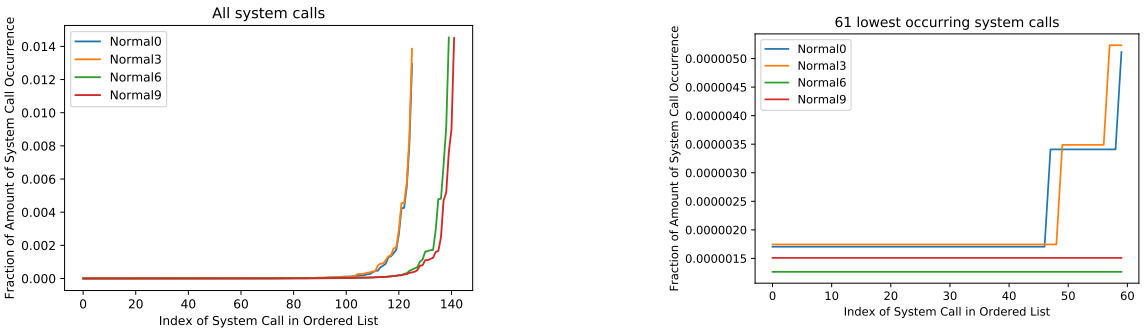

Figure 5.7: Distribution of system call occurrences in benign phases

Continuing, in the original classification, all but one of the evaluated benign system states were added to the attack phase classification task. Next, a newly built attack/benign phase classification is analyzed to show if the classification would also be successful with more (benign) classes. This approach leads to an F1-score of 93.16%. The confusion matrix that

entails 22 classes is shown in Table 5.4 and illustrates that the most confusion happens within the benign system phase classes (Normal0 to Normal10). Further, the macro F1-scores of malware detection respectively anomaly detection of 98.1% respectively 99.4% indicate that increasing the number of training classes only slightly decreases the overall classification results.

		Predicted class																				F1 over 22 classes	F1 over 5 classes	F1 over 2 classes
		Normal0	Normal1	Normal2	Normal3	Normal4	Normal5	Normal6	Normal7	Normal8	Normal9	Normal10	Basilite_Botconnect	Basilite_HOLD	Basilite_UDP	Basilite_TCP	BtvLinstallKdcmake	BtvLb64	BtvLsshLlogin	BtvLattackLloop	RansomwarePoC	Thetick_infection	ThetickLattacks			
Actual class	Normal0	83	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.862		
	Normal1	0	59	11	0	0	1	10	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0.787		
	Normal2	0	0	84	1	0	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.923		
	Normal3	0	0	0	84	0	1	0	0	2	0	0	0	0	0	0	0	0	0	1	0	0	0	0.938		
	Normal4	0	3	0	0	67	8	0	1	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0.905	0.994	0.994
	Normal5	0	1	0	5	0	71	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.830		
	Normal6	0	0	2	0	0	0	79	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.878		
	Normal7	0	0	2	0	0	0	0	98	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0.970		
	Normal8	0	1	0	1	0	0	8	0	87	0	0	0	0	0	0	0	0	0	0	0	0	0	0.916		
	Normal9	0	0	0	0	0	0	1	0	0	81	0	0	0	0	0	1	0	0	0	0	0	0	0.988		
	Normal10	0	1	0	0	0	0	0	0	0	0	77	0	0	0	0	1	0	0	0	0	0	0	0.987		
	Basilite_Botconnect	0	0	0	0	0	0	0	0	0	0	0	77	15	1	0	0	0	0	0	0	1	0	0.890		
	Basilite_HOLD	0	0	0	0	0	0	0	0	0	0	0	0	98	0	0	0	0	0	0	2	1	0	0.891	0.977	
	Basilite_UDP	0	0	0	0	0	0	0	0	0	0	0	0	1	86	0	0	0	0	0	0	0	0	0.989		
	Basilite_TCP	3	0	0	0	0	0	0	0	0	0	2	0	0	0	77	0	0	0	0	0	0	0	0.951		
	BtvLinstallKdcmake	0	0	0	0	0	2	1	0	0	0	0	0	0	0	0	71	11	0	0	0	0	0	0.893	0.989	0.994
	BtvLb64	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	94	0	0	0	0	0	0.935		
	BtvLsshLlogin	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	90	0	0	0	0	1.000		
	BtvLattackLloop	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	89	0	0	0	0.989		
	RansomwarePoC	0	0	0	0	0	0	0	0	0	0	0	2	3	0	0	0	0	0	0	77	0	0	0.957	0.957	
	Thetick_infection	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	76	0	0.974	0.988	
	ThetickLattacks	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	83	1.000		

Table 5.4: Confusion matrix of Neural Network 1-gram & TF-DF of attack phase classification with eleven benign states

5.4 Results with adversarial attacks

Adversarial attacks on machine learning models can happen during the training or test phase of the model. When attacking during the training phase an attacker often tries to alter the dataset such that the classification is still good but the model is trained on data that is different from data in the real world. In image-based classification, this can be done by adding pixels to the pictures which can't be seen by a human eye. Equivalently, unnecessary system calls could be added to the dataset. Despite not being recognizable by the eye, in the case of image classification data could be checked for integrity by comparing the pixels of the original images with the ones that are in the training set. Similarly, before distributing the AI for malware detection it needs to be checked that the underlying data comes from unaltered data. Another attack during training is called *logic corruption* where the parameters of a model are changed such that the classification is less successful. In difference to the latter attack, this attack could be detected when looking at the classification performance of the model. The problem therein is that when the model is used in the field it is cumbersome to check for model performances.

Attacks while testing or in the field are called *evasion attacks*. In such attacks, mostly testing data is modified. In the scenario of malware detection based on system calls, the testing data comes directly from the behavior of a system. In the case of malware detection based on system calls, such an attack could be performed by first observing and collecting system calls from the benign system state and then replicating these system calls into the malware. As learned in the background chapter, Unix systems provide for nearly every system call a corresponding function in the C programming language. Therefore, it can be argued that such adversarial attacks are possible.

The results of the attack that was introduced in 3.1.4 where system calls unique to the benign system state were added to the malware samples can be seen in the row *Unique normal Syscalls* in table 5.5. Such an attack was simulated by taking the TF-(I)DF values of the system calls that are only occurring in the normal dataset and adding these to the TF-(I)DF arrays of the malware samples. The TF-(I)DF values were computed by simulating a random choice of the occurrences of those system calls in the sequences based on their probability distribution which were calculated based on the fraction of how often they occur in the original dataset (for more details see pseudo code in section 4.5.3). This approach was executed with the TF-DF and TF-IDF preprocessing approach and the Neural Network model receiving unigrams as input. In Table 5.5 the classification results of the standard scores and the adversarial scores show that the F1-score stay more or less constant across both preprocessing techniques.

After performing the adversarial attack mentioned above, one could argue that the NN model combined with TF-(I)DF is resilient against such an attack but unfortunately, there exist other attacks that are harder to defend. The next adversarial attack that was carried out, is again based on the distribution of the benign system calls but now includes not only the system calls that are unique to the benign state. In this attack simply the array of the TF-(I)DF values from benign test data was taken and then added to the TF-(I)DF arrays of the attack phases. The results (see Adversarial Attack *Full copy of normal syscalls* in Table 5.5 show that TF-DF is not robust against such an attack as its classification scores

Adversarial Attack	Preprocessing method	
	TF-IDF	TF-DF
None	F1: 53.33%	F1: 99.04%
Unique normal Syscalls	F1: 41.86%	F1: 99.03%
Full copy of normal syscalls	F1: 42.37%	F1: 21.37%
75_percent_normal	F1: 50.63%	F1: 18.48%
50_percent_normal	F1: 53.95%	F1: 19.97%
50%_normal/50%_malware	F1: 33.99%	F1: 8.53%

Table 5.5: Overview of the results of classification with adversarial attacks with keeping the model constant (Neural Network)

decline drastically. Continuing, further scenarios are added. Let's consider two attack cases consisting of a similar attack as the latter but where instead of the full values of the TF-(I)DF arrays only fractions of 50 and 75 percent of the values of the benign feature array are added. In table 5.5 their results are shown with the names *50_percent_normal* respectively *75_percent_normal*. A further experiment was done by taking the TF-(I)DF values from benign and malware samples and multiplying both by 0.5 trying to show what happens when half data comes from malware and the other half from benign system calls (see *50%_normal/50%_malware*).

Although, we can see that TF-DF and to some extent also TF-IDF can't handle these changes in the data the results must be put in perspective. The attacks apart from the one where unique system calls are added cannot be easily performed. Although, carrying out an attack that consists of producing system call sequences that consist of benign and malware system calls with a 50/50 ratio could be possible this wouldn't halve the underlying TF-(I)DF values as the TF-(I)DF value also consists of other factors than the term frequency. Therefore, such attacks would only be possible if an attacker could change the active classifier respectively the code that handles the input system call traces and transforms them into TF-(I)DF arrays. When considering the more probable adversarial attack that includes system calls that are unique to the benign system state the table shows that the TF-DF preprocessing technique is more stable against such an attack in comparison to the TF-IDF method.

5.5 Resource consumption

Table 5.5 shows the time consumption of training and testing the different models. The time consumption was measured on a 12-core Linux server with 98 Gigabytes of RAM. Not only because different models used different kinds of preprocessing approaches but also because they have been evaluated with different amounts of training strategies the time consumption is compared based on the best parameters of each model. As this

	NN	AE	CNN	LSTM	Transformer
Data preprocessing time (in sec)	0.004356	0.01983	0.00941	0.00466	0.03
Model Loading time (in sec)	0.00856	0.00653	0.05615	0.452	1.1552
Classification process time (in sec)	0.000096	0.000189	0.000109	0.000077	0.00055
Total testing time of 10seconds system call trace (in sec)	0.013	0.027	0.066	0.457	1.186
Training time (in min)	59.72	40.3	18.84	45.65	50.22

Table 5.6: Overview over time consumption of different models with the same preprocessing technique TF-DF

doesn't yield a perfect comparison the number of datapoints and the number of batches per model are also indicated in the table.

5.6 Results of equal length input sequences

Regarding the TF-(I)DF calculation in the former sections it was made use of different length input sequences. As through this the results of TF-(I)DF might be misleading some experiments now some experiments are evaluated where the input system call sequences have been of the same length. Table 5.7 illustrates the classification results of whereas the tested hyper-parameters are listed. As indicated, surprisingly here the TF-DF is outperformed by TF-IDF preprocessing technique.

Machine learning model	NN		AE	CNN	LSTM	Transformer
Preprocessing technique	TF-DF	TF-IDF	TF-IDF	TF-IDF	TF-IDF	TF-IDF
Momentum	0,0.5,0.8,**0.9**,0.99,1	0,0.5,0.8,0.9,**0.99**,1	1	0.99	0.99	0.8
Number of sequences (TF-(I)DF)	400					
Learning rate	**0.1**,0.05,0.01,0.001,0.0001	0.1,**0.05**,0.01,0.001,0.0001	0.001	0.1	0.1	0.01
Sequence Length / Batch Size	N.A.	N.A.	N.A.	**10**,20,30,40	**10**,20,30	10
Hidden dimension	10	10	50	10	700	700
F1-score (in %)	46.17	92.9	19	4.79	2.17	0.98

Table 5.7: Results overview where input sequences are balanced with 1-gram system calls (best configurations marked bold)

Chapter 6

Discussion

As introduced in the "Evaluation" chapter, sequence models (CNN, LSTM, Transformer) perform unexpectedly badly when referring to attack phase classification based on system calls. Therefore, one could argue that either the sequence models didn't learn the sequence information well enough or that sequence information is not as important for system call traces as it can be in Natural Language. Vice versa, the distribution of irregular system calls in the evaluated part of the dataset indicates that the sequence models might have learned the wrong sequences. Another reason for this outcome could be that the sequence models have been trained with a smaller amount of training steps, although experiments with a similar amount of training steps have shown no significant performance increase. Further, the training of the sequence models does not include the whole data per training step. Continuing, batch processing might be the bigger driver for the low-performance results as in batch processing after evaluating only a small portion of data the loss and the matrices are updated. Especially, the former might be crucial as this could lead to not finding a minimum in the gradient, although momentum is used to prevent the gradient from going up and down. Another reason could be that bigger models tendentially need more data but also more time to increase their performance [47].

It seems that sequence models perform moderately when being trained with fewer classes which were evaluated shortly with the greedy method where only the initial attacks of each malware and the benign case were used as the base for malware classification. There, the LSTM model with one-hot-encoded unigrams and a sequence length of 10 yielded an accuracy of 52.88 percent. Still, the model has been unable to cope with more diverse data and more classes. Therefore, it is advised to work with a simple Neural Network in system call based malware detection as a malware detection program should be able to handle diverse system call data. When training the network with arrays of TF-(I)DF consisting of values stemming from sequences with different lengths the TF-DF outperforms the TF-IDF preprocessing technique whereas when it is done with equal length sequences the TF-IDF outweighs the TF-DF preprocessing method. In the following two sub-sections, the results of the different preprocessing techniques stemming from the former (unbalanced) method are discussed. After that, the results of the balanced method are discussed while starting to compare them to related work.

6.1 TF-IDF vs. TF-DF

The classification results have shown differences when comparing the two related preprocessing techniques TF-IDF and TF-DF. Interestingly, the NN model seems to have problems with detecting attack phases when using TF-IDF although in related work it is a well-established preprocessing technique. This could stem from the fact that in this thesis the TF-(I)DF values of context system calls have been given to the models in the same fashion as the values from regular system calls whereas other works haven't made use of context system calls as additional features. Adding up the context system calls with the regular system calls of the same system call might lead to a better representation of the TF-IDF curve. However, Figure 6.1 shows that this distribution doesn't show a big difference to the distribution where regular and context system calls have been handled uniquely.

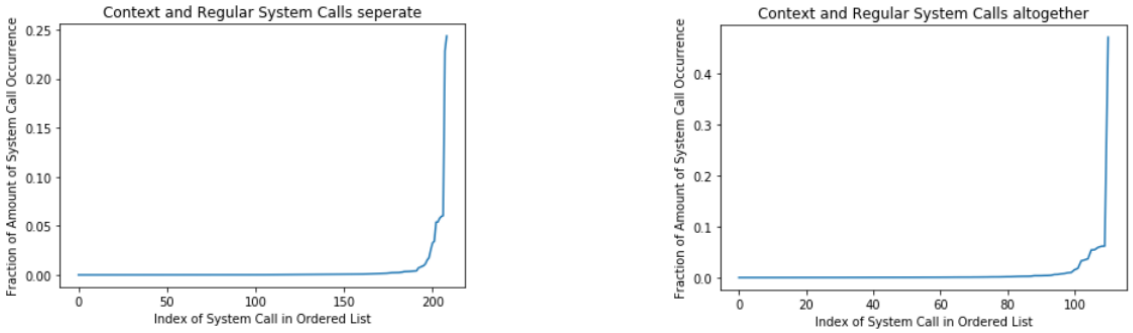

Figure 6.1: Comparison of system call occurrence distributions between original approach where regular system calls and context calls have been summed up together (right) and where they have been summed up together (left)

An explanation of why the performance differences between the TF-IDF and TF-DF data preparation methods are big could lie in the nature of the data. *Term Frequency-Document Inverse Frequency (TF-IDF)* was originally used as the name suggests in documents that consist of Natural Language which can include thousands of different words where some words can have several meanings depending on their contexts, e.g. *bank* account and river *bank*. Although there also exist system calls with the same name having different functionalities, the amount of different system calls reaches only 209 in this analysis, whereas the English dictionary consists of hundreds of thousands of words. As seen in Figure 4.9 (Figure log) TF-DF on one hand, has the most variance between one and ten occurrences of a system call in sequences of the attack phase datasets. After that, the differences between the output values of the DF-function sink drastically. TF-IDF on the other hand has a softer slope at first, reaches its change from decline to incline at around 100 occurrences, and has a constant gradient. This leads to the assumption that TF-IDF better represents general distribution variance whereas TF-DF focuses on distributions of low-occurring system calls. Therefore, the results indicate that in attack phase classification rare system calls are more expressive for a given system state.

Following it is further elaborated on the distribution of the given data and its implications on the TF-DF values. With the chosen preprocessing technique where context system calls

have been equally handled as being normal system calls there have been a total of 209 different system calls where the median system call (clone*) has occurred 29 times in 216'096 system calls. Further, the TF-(I)DF values have been computed based on the sequences of specific attack phases datasets' which had 18'008 system calls in it which results in an average occurrence of the system call clone* of 2.42 times per attack phase. For this distribution, the probability to occur in more than 10 sequences of an attack phase lies between 0.000167% and 0.438% depending on the chosen amount of sequences (nrseqs) and the sequence length (seq_len) (see *Implementation Details*). The probability of clone* being part of more than 10 sequences is computed with the help of the probability x that it occurs in a sequence as follows:

$$x = 1 - (1 - distribution\_of\_clone^*)^{seq\_len}$$

$$P_{in\_seqs}(clone^* > 10) = 1 - \sum_{i=0}^{10}[x^n \cdot \binom{nrseqs}{n} \cdot (1-x)^{nrseqs-n}]$$

Even more, this probability computation method disregards the deduction of the probability of x after clone* has been chosen once or a couple of times the true probability would be even lower. Therefore, it is suggested that around half of the system calls that occur rarely in more than 10 sequences and are therefore well represented by the TF-DF curve. For system calls occurring 10 times more than the median system call which applies to the 175th lowest occurring system call *ugetrlimit* of in total 209 different system calls the probability of its occurring in more than 10 system call sequences rises to 99.99%. Still, those system calls are represented distinguishably by the TF-DF function as the chance of occurring more often than 100 times (for nrseqs=1000) is only 3.33 * 10^{12} percent. Despite the fact, that the variety decreases when the amount of occurrences is in between 10^1 and 10^2 in comparison to the range of 10^0 and 10^1 differences between the number of occurrences are still distinguishable. Finally, Figure 6.2 illustrates the little differences between the number of occurrences until the 175th system call least occurring system call and even further by showing the fractional occurrences of the system calls that are ordered from least occurring to most occurring.

6.2 TF vs. TF-(I)DF

The fact that especially Document Frequency (DF) is nearly a constant factor as it changes its value only at the fourth decimal place when occurring in many documents could lead to the assumption that the Term Frequency by itself is a good enough feature for an ML model as it would behave like the TF-DF value where DF is constant (=1). This hypothesis has been rejected in a greedy malware classification experiment where the model yielded an accuracy of 46.4% where sequences of unigrams have been preprocessed into arrays of term frequency values while being optimized with the limited amount of hyper-parameters (e.g. number of sequences). Concurrently, TF-IDF outperformed the term frequency preprocessing method yielding an accuracy of 86.47%.

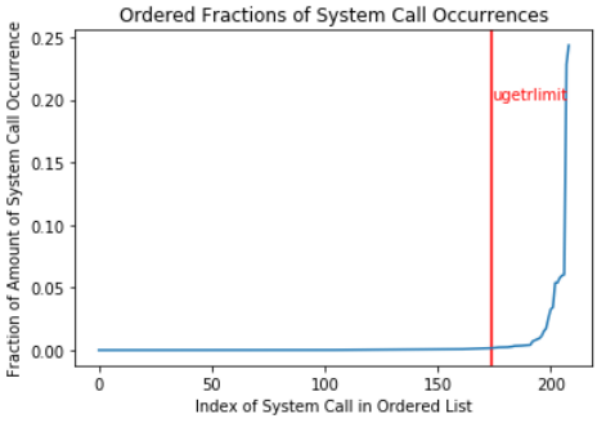

Figure 6.2: Ordered (from least to most occurring) system call occurrences where 175th least occurring system call *ugetrlimit* is marked

6.3 Comparison of results with related work

The authors of [9] also have conducted a multi-class classification where different attack phases are classified with rather simple Machine Learning models, such as the Random Forest or Support Vector Machine models, whereas their input features stemmed from monitoring various resource usage types including events collections such as tcp or udp for network usage or measures of CPU usage (including rpm, ipi, and clk). Having more distinct classes they yielded an F1-score of 96 percent and with that outperformed the simple Neural Network implemented in the work of this thesis when the model of this thesis receives a similar amount of classes as input (see 5.3). However, a part of the better performance could be explained due to the proportional bigger amount of similar benign phase input data in this thesis. Another reason might be the big amount of features used in the mentioned work which can make a model more robust for classifying similar system states.

In difference to the related work, in this thesis the sequence models have not been combined as the benefits of this seem to be minimal. The authors of [42] e.g. tried to combine a Convolutional Neural Network with a Long-Short Term Memory model which only resulted in a 0.2% accuracy increasement in comparison with having a Convolutional Neural Network alone. Equivalently, in the work [22] a Transformer model has been combined with an LSTM model. This resulted in the same performance increase (0.2%) whereas it was measured in precision. Due to these minimal improvements the results of this work are solely compared with the works that use single models.

Whereas the work [42] yielded good results with the CNN the results of the CNN model implemented in this thesis are weak when looking at the attack phase classification. Although, both classification problems have a similar amount of classes a reason for the drastic performance differences might stem from the differing detection tasks. Whereas the mentioned work tried to classify malware from different malware families, in this work the focus was put on detecting different attack phases. Another reason for the performance variations might come from the underlying dataset that is structured differently. Whereas in this work the context system calls were treated as additional system calls

without connection to the system call category, in the mentioned work system calls and context system calls weren't differentiated at all. Furthermore, the already stated amount of training steps and limited hyper-parameter tuning could also have had effects on the models' performance. A further difference is found in the preprocessing whereas in this work one-hot-encoding is performed on 1-grams of system calls without further modification of the data the authors of [42] remove identical API calls when they arise more than twice one after another. However, such an experiment with the underlying dataset was performed and didn't outperform the original result.

Finally, revisiting the distribution of (see Figure 6.3) irregular system calls concerning the system calls of the evaluated data (*NormalBenign10s*) used in this thesis could be a further cause for the mal-performance of the sequence models, especially when using one-hot-encoding as preprocessing technique. Since the Figure illustrates that some classes have a high amount of irregular system calls it is indicated that system calls are quite often ordered incorrectly which potentially misleads a sequence model. Still, such irregularities might also occur in the field. Therefore, a model must be able to also handle unordered system call sequences.

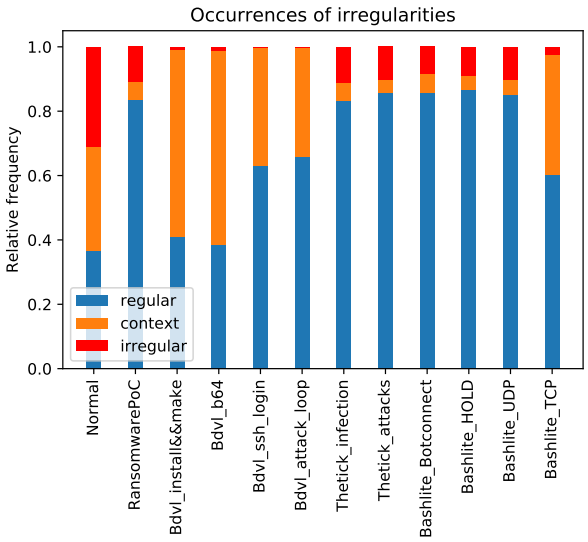

Figure 6.3: Distribution of irregular, context, and regular system calls over different attack types within *NormalBenign10s* dataset

6.4 Robustness against adversarial attacks

Security also needs to be considered when implementing Artificial Intelligence. Attackers can tamper with the prediction of Machine Learning models by introducing adversarial samples. In the case of system call sequences, adversarial samples would correspond to extra system calls that try to obfuscate that an attack is happening. Such an attack would need a moderate understanding of how to install unnecessary system calls into existing or new malware.

On one hand, the experiments with adversarial attacks have shown that the implemented neural network and the preprocessing technique TF-DF are robust against the unique system call attack which has been based on the implementation of the authors of [4]. On the other hand, when executing an adversarial attack that not only takes unique benign system calls but takes all system call occurrences and translates the term frequencies of the system calls into TF-(I)DF arrays and adds those to the already existing TF-(I)DF malware arrays the Neural Network can be misled. The former adversarial attack method would need knowledge of the distribution of the system calls of the benign dataset or the benign state of the target system. Although an attacker could access the benign state or dataset, these attacks still might be hard to execute as malware might hinder the system from performing these benign system calls. Also, a real-time attack could be less harmful if it is done based on real-time data not stemming from the dataset the model was trained with. Additionally, the attacks where the TF-(I)DF values are directly altered are less straightforward because an attacker would first need to find the code where the live system calls are split into sequences and then transformed into TF-(I)DF arrays and change the code as presented in the beforehand chapter. Still, the possible unrobustness of malware detection algorithms needs to be addressed by hiding the code that processes field data.

Further, it is advised to use a system call based malware detection system in combination with a model that uses other behavioral features such as changes in usage of CPU and/or memory as used in the work [9] where many behavioral features were used. Those make adversarial attacks more difficult because it is very unlikely that an attacker can find out which feature is the most influential for the model's classification. And even if, the feature might be hard to influence. Still, for companies or private persons that can't have both protection mechanisms, it could be elaborated if given system call based malware detection models perform similarly by introducing benign system calls into the malware system call samples without reducing the performance of the model to a relevant extent. Another way of preventing adversarial attacks could be found when including additional features such as the return messages of system calls, the time they are active, and a boolean indicating if the system call is executed in an irregular period.

6.5 Practical considerations

Implementing a model which is combined with a TF-IDF or TF-DF preprocessing needs to take into consideration that in the field the preprocessing would need to be handled similarly. TF-(I)DF methods need to look at multiple sequences first before being able to provide a reasonable TF-(I)Df value. In this work, the TF-(I)DF values of each attack were calculated based on 18'008 system calls. Depending on the attack phase the time that passed between the first and the last system call in between the whole training and test data spans from between 0.28 seconds in the benign state and 10 seconds in the *Bashlite_Hold* attack phase. Therefore, the question arises of how long a system needs to collect data before classifying the data. E.g. if orientating on the attacks where the system takes the longest time to produce 18'008 system calls one would need to wait for ten seconds before being able to classify the current data. In this case, one would need to enable the parallelization of preprocessing and classifying data when wanting to be able

to classify the system's behaviour in a faster fashion. If no parallelization is possible due to scarce system resources one might want to train the model with less data although that might weaken its performance.

6.6 Limitations

While the model performances' of unbalanced sequence lengths have been evaluated elaborately results for balanced TF-(I)DF calculation are scarce. Beyond this, the overall good results of the neural network model are only based on one dataset and therefore, the overall performance of the implemented Neural Network and Autoencoder need to be further justified. Since the models' performances haven't been evaluated when attack phases of related malware families need to be distinguished. Neither, are the models' behaviors known when including different malware types such as Worms or Viruses. Another limitation is the number of training steps performed in the sequence models which has been limited to 100 epochs due to time constraints. Furthermore, memory constraints of the used training system impeded the evaluation of the sequence models' performance with higher numbered n-grams (n>1).

6.7 Future Steps

As a next step it is proposed to broaden the performance analyses of the balanced input sequences together with TF-IDF preprocessing technique. This can be achieved by simply increasing the amount of parameters to the ones seen in chapter *Implementation Details*. Further, for some sequence models the evaluation must be repeated as proposed in the mentioned chapter.

Probably the most promising direction is to further evaluating the performance of the Neural Network and the Autoencoder model combined with different n-grams but also with broader datasets as indicated before. Therefore, producing more data or applying the models to unknown system call sequences would be necessary. In the latter strategy, the accuracy of the classification potentially decreases. Therefore, one would have to adapt the model to enable handling the unknown malware system call sequences. Unfortunately, one can't simply threshold the confidence values of the output layer and let that decide on the new data being part of the unknown class or not, because the confidence value can't be seen as an absolute certainty value but is relative to the input and the similarity of classes. This brings us to the open set problem is an unresolved issue in machine learning. It can be described as a model being unable to classify data that is part of a class that is not contained in the training data. The similarity of classes was used by the authors of [7] as a suggestion to solve the open set problem by calculating an additional output neuron to the output layer based on the distances between the values of the original output layer. The advantage of this approach is that there is no need for additional unknown data in the training phase. Still, it would be needed to evaluate if this approach also works for the dataset studied in this thesis. Producing more data alone would increase the

accuracy but might only alleviate the open set problem datasets in the attacking realm as totally new attacks including e.g. a *zero-day attack* could arise. Another reason why introducing an unknown malware class could make sense is that misclassifications could be prevented when malware of other malware families are classified. Possibly, generative networks could be used to produce data for the unknown malware class. Further, such an approach could also increase the stability of the model against adversarial attacks. Concluding, evaluating the open set problem can be key for the model to recognize system state anomalies stemming from unknown malware classes and/or newly created malware programs.

The second limitation would need to be addressed by increasing the number of epochs to 1'000 for the sequence models in which case the number of total training steps would be balanced out with the total number of epochs of 10'000 in the Neural Network and the Autoencoder model. However, some first experiments with the same amount of training steps in all models have shown no major performance improvements. But, this might not be enough as the authors of [47] indicate that Deep Learning models such as CNNs, ResNets, and Transformers underly the Double-Descent phenomenon which means that they generally need an extensive amount of training time to reach good performance results respectively e.g. finding the global minimum of the gradient. Therefore, the sequence models would need to be evaluated with an even higher amount of training steps to find the best performance results of these models. Furthermore, another way the limited performances of the sequence models could be addressed is by reordering the irregular system calls and reevaluating the models' performances. This is expected to at least improve the results of classifying sequences of one-hot-encoded system calls. Finally, running the sequence models with higher n-grams on a machine with an extensive size of RAM could improve the classification of preprocessed sequences of system calls by both one-hot-encodings and TF-(I)DF transformations.

Chapter 7

Summary and Conclusions

Altogether, several models with different hyper-parameter combinations have been evaluated to do attack phase classification and detection based on system calls. The dataset building the foundation of this thesis contained different cleaned system call traces that have been collected while the system has been benign or infected by four different malware programs. Although, some attack/installation phases only comprise less than twenty thousand system calls the NN model has been able to successfully classify not only the malware type but also the attack phases of the evaluated malware types.

The model with the highest performance but also with the fastest training time was the Neural Network with the *TF-DF* (Term Frequency Document Frequency) preprocessing technique when using imbalanced input data. The worse performance of the sequence models has been widely discussed, whereas different potential factors have been depicted such as: the limited number of training steps, standard vs. inferior batch processing, and dataset irregularities. Further, the amount and characteristics of classes of attack phase classification and the novel way of including context system calls as unique features might have impeded the performance of the sequence models. Another aspect of this thesis was to better capture the distribution of expressive system calls with the help of an adapted preprocessing technique *TF-DF* based on the well-known TF-IDF method to better capture the distribution of the system calls occurring in the dataset. The proposed *TF-DF* seems to outperform the TF-IDF preprocessing approach used in Natural Language Processing that also has already been successfully used for anomaly and malware detection based on system calls when serving the model with unbalanced input data. Whereas the classification experiments and some data analysis have shown that *TF-DF* maps the data distribution of the underlying system calls better than the well-known TF-IDF preprocessing technique introducing balanced sequence lengths has shown that a result in another direction.

When training the Neural Network with preprocessing technique *TF-DF* and 1-gram combined with the best-fitting hyperparameters stemming from imbalanced input sequences yielded an F1-score of 99.2 percent whereas when using balanced data the same model with the *TF-IDF* preprocessing technique reached an F1-score of 92.9 percent. When reducing the former result from imbalanced attack phase classification to an anomaly classification an accuracy of 100% was achieved. Therefore, the goal of this work to have malware

detection or attack classification while in parallel evaluating if the system is behaving anomalously or not was reached. Even more, the model seems to be able to differentiate between different attack phases of the malware. Still, the model's performance evaluating unknown malware attack samples and measures to keep the stability against adversarial attacks need to be discussed. In this direction, it could help to have additional features of the system's behavior included in a malware detection program such as CPU or memory usage. Further, it was discussed that the non-existence of installation procedures in the normal data could be the reason why in the best-performing model the benign case was never confused with an attack phase. At the same time, in IoT devices, it is not unusual to have such behaviour as the devices are not under active use and therefore some devices' most similar activities to installations are limited to security patches and other updates which might neither be confused with the installations of malware.

Concluding, the good performances of a few models seem to be promising results some further considerations have to be made. As seen, some first experiments with adversarial attacks have shown that they can to some degree harm the detection of attacks. Continuing, the well-performing models would need to be tested on different datasets and with malware types and families that haven't been evaluated in this work yet.

Bibliography

[1] Mohammed Al-Asli and Taher Ahmed Ghaleb. Review of signature-based techniques in antivirus products. In *2019 International Conference on Computer and Information Sciences (ICCIS)*, pages 1–6. IEEE, 2019.

[2] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *Proceedings of 2017 International Conference on Engineering and Technology, ICET 2017*, volume 2018-January, pages 1–6. Institute of Electrical and Electronics Engineers Inc., 3 2018.

[3] Ni An, Alexander Duff, Gaurav Naik, Michalis Faloutsos, Steven Weber, and Spiros Mancoridis. Behavioral Anomaly Detection of Malware on }Home Routers. In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 47–54. IEEE, 2017.

[4] A. Ananya, A. Aswathy, T. R. Amal, P. G. Swathy, P. Vinod, and Shojafar Mohammad. SysDroid: a dynamic ML-based android malware analyzer using system call traces. *Cluster Computing*, 23(4):2789–2808, 12 2020.

[5] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Ndss*, pages 23–26. Ndss, 2014.

[6] K. A. Asmitha and P. Vinod. A machine learning approach for linux malware detection. In *Proceedings of the 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques, ICICT 2014*, pages 825–830. IEEE Computer Society, 2014.

[7] Abhijit Bendale and Terrance E Boult. Towards Open Set Deep Networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1563–1572, 2016.

[8] Daniel Pierre Bovet and Marco Cesati. *Understanding the Linux Kernel*. OReilly Media, 2005.

[9] Alberto Huertas Celdrán, Pedro Miguel Sánchez Sánchez, Miguel Azorín Castillo, Gérôme Bovet, Gregorio Martínez Pérez, and Burkhard Stiller. Intelligent and behavioral-based detection of malware in IoT spectrum sensors. *International Journal of Information Security*, pages 1–21, 7 2022.

[10] Ping Chen, Lieven Desmet, and Christophe Huygens. A study on Advanced Persistent Threats. In *IFIP International Conference on Communications and Multimedia Security*, pages 63–72. Springer, Berlin, 2014.

[11] deadPix3l. CryptSky, 6 2020.

[12] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 3 2016.

[13] Michael Dymshits, Benjamin Myara, and David Tolpin. Process Monitoring on Sequences of System Call Count Vectors. In *2017 International Carnahan Conference on Security Technology (ICCST)*, pages 1–5, Madrid, 10 2017. Carnaham.

[14] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[15] Error996. bdvl, 2 2022.

[16] Maryam Feily, Alireza Shahrestani, and Sureswaran Ramadass. A survey of botnet and botnet detection. In *Proceedings - 2009 3rd International Conference on Emerging Security Information, Systems and Technologies, SECURWARE 2009*, pages 268–273, 2009.

[17] Chao Feng. *Intelligent Analysis of System Calls to Detect Cyber Attacks Affecting Spectrum Data Integrity in IoT Sensors*. PhD thesis, University of Zurich, Zurich, 1 2022.

[18] David Forsyth. *Probability and Statistics for Computer Science*. Springer, 2018.

[19] Susan E Gathercole. *Models of short-term memory*. Psychology Press, 2013.

[20] Yoav Goldberg. *Neural Network Methods for Natural Language Processing*. Morgan & Claypool Publishers, 2017.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[22] Yue Guan and Naser Ezzati-Jivan. Malware System Calls Detection Using Hybrid System. In *15th Annual IEEE International Systems Conference, SysCon 2021 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., 4 2021.

[23] Soheil Hashemi and Mani Zarei. Internet of Things backdoors: resource management issues, security challenges, and detection methods. *Transactions on Emerging Telecommunications Technologies*, 32(2):e4142, 2021.

[24] Md Jobair Hossain Faruk, Hossain Shahriar, Maria Valero, Farhat Lamia Barsha, Shahriar Sobhan, Md Abdullah Khan, Michael Whitman, Alfredo Cuzzocrea, Dan Lo, Akond Rahman, and Fan Wu. Malware Detection and Prevention using Artificial Intelligence Techniques. In *2021 IEEE International Conference on Big Data*, pages 5369–5377. Institute of Electrical and Electronics Engineers Inc., 2021.

[25] Yuming Hua, Junhai Guo, and Hua Zhao. Deep Belief Networks and deep learning. In *Proceedings of 2015 International Conference on Intelligent Computing and Internet of Things, ICIT 2015*, pages 1–4. Institute of Electrical and Electronics Engineers Inc., 5 2015.

[26] Yaqoob Ibrar, Ahmed Ejaz, Muhammad Habib ur Rehman, Ahmed Abdelmuttlib Ibrahim Abdalla, Mohammed Ali Al-garadi, Imran Muhammad, and Guizani Mohsen. The rise of ransomware and emerging security challenges in the Internet of Things. *Computer Networks*, 129:444–458, 2017.

[27] IoT Analytics. State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally.

[28] M. Asha Jerlin and K. Marimuthu. A New Malware Detection System Using Machine Learning Techniques for API Call Sequences. *Journal of Applied Security Research*, 13(1):45–62, 1 2018.

[29] jimmy-ly00. Ransomware-PoC, 3 2021.

[30] Michael Kerrisk. getpgrp(3p) — Linux manual page, 2017.

[31] Michael Kerrisk. sendmmsg(2) — Linux manual page, 6 2020.

[32] Michael Kerrisk. getdents(2) — Linux manual page, 3 2021.

[33] Michael Kerrisk. perf-trace(1) — Linux manual page, 3 2021.

[34] Michael Kerrisk. setgid(2) — Linux manual page, 3 2021.

[35] Michael Kerrisk. setpgid(2) — Linux manual page, 3 2021.

[36] Michael Kerrisk. symlink(2) — Linux manual page, 8 2021.

[37] Michael Kerrisk. unlink(2) — Linux manual page, 8 2021.

[38] Minhaj Ahmad Khan and Khaled Salah. IoT security: Review, blockchain solutions, and open challenges. *Future Generation Computer Systems*, 82:395–411, 2018.

[39] Dang Kien, Hoang Dai, Tho Nguyen, and Duy Loi Vu. IoT Malware Classification Based on System Calls. Technical report, 2020.

[40] Chan Woo Kim. NtMalDetect: A Machine Learning Approach to Malware Detection Using Native API System Calls. 2 2018.

[41] S H Kok, Azween Abdullah, N Z Jhanjhi, and Mahadevan Supramaniam. Ransomware, Threat and Detection Techniques: A Review. Technical Report 2, 2019.

[42] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep Learning for Classification of Malware System Call Sequences. In *Australasian joint conference on artificial intelligence*, pages 137–149. Springer, 2016.

[43] Arash Habibi Lashkari, Andi Fitriah A. Kadir, Laya Taheri, and Ali A. Ghorbani. Toward Developing a Systematic Approach to
}Generate Benchmark Android Malware Datasets
}and Classification. In *52nd Annual IEEE International Carnahan Conference on Security Technology (ICCST)*, pages 1–7, Montréal, 2018.

[44] Ming Liu, Zhi Xue, Xianghua Xu, Changmin Zhong, and Jinjun Chen. Host-based intrusion detection system with system calls: Review and future trends, 1 2019.

[45] Artur Marzano, David Alexander, Osvaldo Fonseca, Elverton Fazzion, Cristine Hoepers, Klaus Steding-Jessen, Marcelo H. P. C. Chaves, Ítalo Cunha, Dorgival Guedes, and Wagner Meira. The Evolution of Bashlite and Mirai IoT Botnets. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 813–818, 2018.

[46] Paul Michel. Are Sixteen Heads Really Better than One?, 3 2020.

[47] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever Openai. Deep double descent: Where bigger models and more data hurt. *Journal of Statistical Mechanics: Theory and Experiment*, 2021(12):124003, 2021.

[48] nccgroup. thetick, 10 2020.

[49] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Łukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image Transformer. In *International conference on machine learning*, pages 4055–4064. PMLR, 2018.

[50] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In David McAllester and Sanjo Dasgupta, editors, *Proceedings of the 30th International Conference on Machine Learning*, pages 1310–1318, Atlanta, Georgia, USA, 6 2013. PMLR.

[51] Naser Peiravian and Xingquan Zhu. Machine learning for Android malware detection using permission and API calls. In *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI*, pages 300–305, 2013.

[52] Sreeraj Rajendran, Roberto Calvo-Palomino, Markus Fuchs, Bertold Van Den Bergh, Hector Cordobes, Domenico Giustiniano, Sofie Pollin, and Vincent Lenders. Electrosense: Open and Big Spectrum Data. *IEEE Communications Magazine*, 56(1):210–217, 1 2018.

[53] Zhongru Ren, Haomin Wu, Qian Ning, Iftikhar Hussain, and Bingcai Chen. End-to-end malware detection for android IoT devices using deep learning. *Ad Hoc Networks*, 101, 4 2020.

[54] Matthew Schofield, Gulsum Alicioglu, Russell Binaco, Paul Turner, Cameron Thatcher, Alex Lam, and Bo Sun. Convolutional Neural Network for Malware Classification Based on API Call Sequence. pages 85–98. Academy and Industry Research Collaboration Center (AIRCC), 1 2021.

[55] M. Shobana and S. Poonkuzhali. A novel approach to detect IoT malware by system }calls using Deep learning techniques. In *2020 International Conference on Innovative Trends in Information Technology (ICITIIT)*, pages 1–5, 2020.

[56] Abraham Silberschatz, Peter B, and Greg Gagne. *Operating system concepts, 10e Abridged Print Companion*. John Wiley & Sons, 2018.

[57] Arpit Singh and Anuradha Purohit. A Survey on Methods for Solving Data Imbalance Problem for Classification. Technical Report 15, 2015.

[58] Ramon Solo de Zaldivar. *Creation of a Dataset Modeling the System Calls of Spectrum Sensors Affected by Malware*. PhD thesis, University of Zurich, Zurich, 4 2022.

[59] Ramon Solo de Zaldivar, Alberto Huertas Celdrán, Jan von der Assen, Pedro Miguel Sánchez Sánchez, Gérôme Bovet, Gregorio Martínez Pérez, and Burkhard Stiller. MalwSpecSys: A Dataset Containing Syscalls of an IoT Spectrum Sensor Affected by Heterogeneous Malware, 5 2022.

[60] Laya Taheri, Andi Fitriah, Abdul Kadir, and Arash Habibi Lashkar. Extensible Android Malware Detection and Family
}Classification Using Network-Flows and API-Calls. 2019.

[61] Greg Van Houdt, Carlos Mosquera, and Gonzalo Nápoles. A review on the long short-term memory model. *Artificial Intelligence Review*, 53(8):5929–5955, 12 2020.

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. 6 2017.

[63] Sun-Chong Wang. *Interdisciplinary computing in Java programming*. Springer, 2003.

[64] Yasi Wang, Hongxun Yao, and Sicheng Zhao. Auto-encoder based dimensionality reduction. *Neurocomputing*, 184:232–242, 2016.

[65] Mohammad Wazid, Ashok Kumar Das, Joel J.P.C. Rodrigues, Sachin Shetty, and Youngho Park. IoMT Malware Detection Approaches: Analysis and Research Challenges. *IEEE Access*, 7:182459–182476, 2019.

[66] Xi Xiao, Shaofeng Zhang, Francesco Mercaldo, Guangwu Hu, and Arun Kumar Sangaiah. Android malware detection based on system call sequences and LSTM. *Multimedia Tools and Applications*, 78(4):3979–3999, 2 2019.

[67] Lifan Xu, Jose Andre Morales, Dongping Zhang, Xudong Ma, Marco A Alvarez, and John Cavazos. Dynamic Android Malware Classification Using Graph-Based Representations. Technical report.

[68] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. DroidDetector: Android Malware Characterization and Detection Using Deep Learning. Technical Report 1, 2016.

[69] Oleg Zaytsev. *Rootkits, spyware/adware, keyloggers and backdoors: detection and neutralization*. -, 2006.

[70] Zhaohui Zheng, Xiaoyun Wu, and Rohini Srihari. Feature Selection for Text Categorization on Imbalanced Data. *ACM Sigkdd Explorations Newsletter*, 6(1):80–89, 2004.

List of Figures

List of Tables