University of Zurich[UZH]

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

Creation of a Dataset Modeling the System Calls of Spectrum Sensors Affected by Malware

Ramon Solo de Zaldivar
Zurich, Switzerland
Student ID: 18-708-552

Supervisor: Dr. Alberto Huertas Celdran, Jan von der Assen
Date of Submission: April 18, 2022

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

ifi

Bachelor 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 Nutzung von IoT-Geräten bringt zahlreiche neue Anwendungsfälle mit sich. Von der Gesundheitsfürsorge über die Standortverfolgung bis hin zu Prozessautomatisierung und Crowdsensing werden IoT-Geräte mehr denn je eingesetzt. Aufgrund dessen wächst die Sorge um die Cybersicherheit, da das System zu einem begehrten Ziel für Cyberangreifer wurde. Durch ihre weitreichende Verwendung können IoT-Geräte Zugang zu einer Vielzahl von Daten haben, was sie zu einem attraktiven Ziel für Cyberkriminelle macht. Diese Geräte bieten keine Unterstützung von herkömmlicher Sicherheitssoftware und sind deshalb nur unzureichend gesichert. Sie sind das Ziel verschiedener Arten von Malware, Botnetzen und Backdoors sowie auch Rootkits und.

Eine praktikable Möglichkeit, diese Cybersicherheitsprobleme zu lösen und gezielte Malware-Angriffe zu verhindern, sind Einbruchmeldesysteme (IDSs). Herkömmliche IDSs sind jedoch nicht fähig neue unbekannte Malware-Angriffe zu erkennen, die auch als Zero-Day-Angriffe bekannt sind. Aus diesem Grund setzen neue Forschungen stark auf Machine Learning (ML) und Deep Learning (DL) basierte IDSs. Eine Schlüsselkomponente, um die Wirksamkeit dieser IDS zu bestimmen, ist ein hochwertiger Datensatz, der das Verhalten eines Geräts bei normalem Verhalten und auch das Verhalten bei neuartiger Komprimittierung durch Malware enthält und mit dem das ML- oder DL-basierte IDS trainiert werden kann. Ein ML- oder DL-basiertes IDS mit einem hochwertigen Datensatz ist statistisch gesehen besser geeignet, neue Malware zu erkennen. Trotz der Bedeutung dieser Datensätze gibt es nur wenige von hâoherer QualitÃ¤t, die das interne Verhalten von IoT-Geräten im Normalzustand und bei Angriffen durch Zero-Day-Angriffe wie Botnets und Backdoors modellieren.

Infolge dieser Einschränkung zielt diese Arbeit darauf ab, einen Qualitätsdatensatz zu erstellen, der das interne Verhalten eines IoT-Geräts sowohl im Normalbetrieb als auch bei einem Angriff genau darstellt. Um dies zu erreichen, werden die Systemaufrufe des IoT-Geräts, in diesem Fall ein ElectroSense-Sensor, bei normalem Verhalten überwacht, gesammelt, bereinigt und in einem zentralen Verzeichnis gespeichert. Anschliessend wird das Gerät mit aktueller Malware infiziert, welche die IoT-Geräte angreift. Hierfür wird ein Bashlite-Botnet, das Thetick-Backdoor, dem Bdvl-Rootkit und einem Ransomware-Proof-of-Concept benutzt, worauf im Anschluss der Überwachungs-, Erfassungs- und Speicherprozess wiederholt wird. Die Infektionen erfolgen sequentiell, d. h. das Gerät wird nicht mit mehr als einer Malware gleichzeitig infiziert. Der generierte Datensatz zeigt normales und anormales Verhalten auf, welches durch die verschiedenen Malwares klassifiziert wird. Abschliessend werden die Sequenzen und Häufigkeiten der Systemaufrufe statistisch ausgewertet.

Abstract

The growing usage of IoT devices brings in itself multiple different new use cases. From healthcare, location tracking to process automations and crowdsensing, IoT devices are being used more than ever. In parallel there has been a growing cybersecurity concern, as IoT devices are becoming a desirable target for cyber attackers. IoT devices, depending on their purpose can have access to large amounts of data which makes them an attractive target for cyber criminals. To further this issue, these devices are poorly secured and inherently, as they are resource constrained, can not support conventional cybersecurity software. IoT devices have been the targets of different kinds of malware, from botnets and backdoors to rootkits, ransomwares and others.

A feasible way to sever these cyber security concerns and prevent these targeted malware attacks from happening, is with the help of Intrusion Detection Systems (IDSs). Nevertheless, traditional IDSs are powerless when it comes to detecting new unknown malware attacks, other wise known as zero day attacks. For this reason, new research is relying heavily on Machine Learning (ML) and Deep Learning (DL) decision engine based IDSs. A key component that determines the efficacy of these IDSs is a quality dataset, containing the behavior of a device under normal behavior and also the behavior when it has been compromised by novel malware, with which the ML or DL based IDS can be trained. A ML or DL based IDS with a quality dataset is then statistically better suited to detect novel malware. In spite of the importance of these datasets, quality datasets, especially ones modelling the internal behavior of IoT devices in a normal state and when under attack by zero day attacks such as botnets, backdoors and others, are scarce.

In wake of this limitation, this thesis aims to create a quality dataset that accurately represents the internal behavior of an IoT device, both when it is functioning normally and when it is under attack. In order to accomplish this, the system calls of the IoT device, which in this specific case is an ElectroSense sensor, are monitored under normal behavior, gathered, cleaned and stored in a centralized directory. Then, the device is infected with current malware affecting IoT devices, such as the bashlite botnet, thetick backdoor, bdvl rootkit and a ransomware proof of concept and the monitoring process is repeated for each malware. The infections are sequential, meaning that the device is not infected with more than one malware at a time. Finally the generated dataset contains normal and anomalous behavior classified by malware. It is then evaluated through analyzing the sequences and frequencies of the system calls statistically.

Acknowledgments

I would like to extend a warm thank you to my supervisors Dr. Alberto Huertas Celdran and Jan Von der Assen for their guidance and insightful inputs during my thesis. Our meetings and discussions gave me a clear perspective of the road map for this thesis.

I would also like to thank and acknowledge the ones closest to me for supporting me unconditionally throughout this thesis.

Further, I would to express my gratitude to Prof. Dr. Burkhard Stiller and the whole Communication Systems Group (CSG) of the University of Zurich for allowing me to complete this challenging and rewarding thesis at their research group.

Contents

Chapter 1

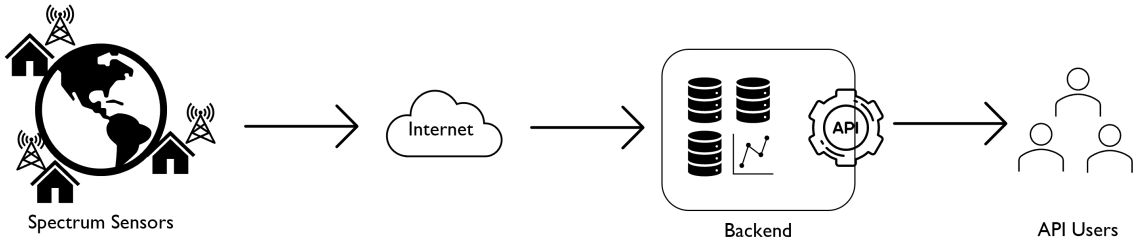
Introduction

1.1 Motivation

The ever-growing demand and usage of Internet-of-Things (IoT) devices are enabling many new and multi faced use cases, such as crowdsensing. Crowdsensing takes advantage of spread out IoT devices to gather data and send it to a crowdsensing platform where the collected data is aggregated, processed and analyzed. At the same time, this growing usage of IoT devices and the large amounts of data being gathered by them, makes these devices more attractive for cyber attackers. Thus, there has been a growing cybersecurity concern affecting the integrity, availability and confidentiality of the sensitive data that is sensed, processed and stored by these IoT devices. In view of the fact that IoT devices are cheap and resource-constrained, they do not hold the computing capacity necessary to support state of the art security software to fend off malicious attacks. This vulnerability can have major negative consequences and monetary implications in case of a data breach [2].

ElectroSense, a crowdsensing platform dedicated to sensing and analysing the radio frequency spectrum, relies on Raspberry Pis attached to a software defined radio (sdr) frontend and an antenna to collect the data [74]. Raspberry Pis are single-board computers that are cheap and can be used as IoT devices [3]. The ElectroSense network consists of these Raspberry Pis acting as sensors, that collect the data which is then sent to the ElectroSense backend. The ElectroSense backend runs different algorithms on the gathered data and offers its results to the users via API [74]. A high level overview of the network is illustrated in Figure 1.1. ElectroSense provides multiple use cases. From retrieving the electrosmog measurements, optimizing indoor networks, to helping governmental entities enforce spectrum regulations [74]. As a consequence of these use cases mentioned above and all the other areas in which IoT devices are being implemented, there is a growing necessity to make these devices as secure as possible. IoT devices, as mentioned before, are naturally not as secure as other devices, due to their simple setup and resource constraints. Further, a report from 2017 [4] stated that multiple IoT devices manufactures release devices with default credentials, some of which are hard coded and can not be

Figure 1.1: ElectroSense Network



changed. These security flaws combined, make IoT devices an attractive target for cybersecurity attacks. For this reason, there is a need to bolster the security of these devices to prevent future compromise.

Traditionally, cybersecurity systems used to detect intrusions provoked by heterogeneous malware, by relying on signature databases. This approach of comparing the signature of a program with the signatures of the malware database has become inefficient, as today, attackers are fully aware of these databases and they purposely modify the malware to ensure that their signatures are different from the ones stored in these databases. These traditional cybersecurity systems are not able to detect new malware variants or families, also known as zero-day attacks as their signatures are not found in the databases [19].

Currently, the trend to avoid this inefficiency has been the usage of classifiers and anomaly detectors based on Machine Learning (ML) and Deep Learning (DL) [8, 9, 10]. Nonetheless, there are still multiple open challenges to create accurate models that are able to detect and classify malware in a timely manner. One of the open challenges is the creation of precise datasets modeling the effects on the internal behavior of resource limited devices while they are under attack from more recent, sophisticated and dangerous malware families. These datasets would then be used to train decision engines that could effectively recognize that a device is under attack in a timely manner [5]. Most of the existing literature and datasets however, dealing with resource-constrained devices and anomalies produced by malware do not model the devices' internal behavior, but rather focus on their network communications, thus, being useless for particular malware families. Some malware families like backdoors, rootkits and ransomware are undetectable over the network, as their exploits compromise a device and not a network. By not creating datasets that revolve around gathering data of the internal behaviour of the devices under attack by these malware, it leaves them even more vulnerable to these types of attacks. This highlights the importance of creating datasets that model the internal behaviour of these resource-constrained devices. An issue that further underlines the importance of these datasets, is that existing datasets are obsolete, since they do not consider recent malware affecting resource-constrained devices. Also, because these IoT devices, in this case more specifically spectrum sensors, can not use current security measures, it is important to have a dataset modelling the internal behavior so they can be then used together with ML or DL to prevent cyberattacks to happen or at least minimize the consequences.

1.2 Description of Work

This work focuses on addressing the challenges outlined above by creating a novel dataset modeling the system calls of a resource-constrained device while infected with different relevant and recent cyberattacks affecting data privacy, availability and integrity. The target device that will be used for testing in this work is a Raspberry Pi, which is part of the ElectroSense IoT crowdsensing network. The system calls of the test device will be monitored while under attack of malware such as backdoors, botnets, rootkits and ransomwares and also while the device is operating normally. Therefore, this thesis involves the following key aspects:

- Research, analyse and select most suitable approach to monitor system calls of a Raspberry Pi running a Linux based OS without impacting the performance of its dedicated function.
- An analysis of different malware families affecting IoT devices. Comparing different malware types such as backdoors, botnets, rootkits and ransomwares and understanding how they work and how they affect the ElectroSense sensor.
- Selection and execution of one vector per malware family on the target device.
- Design, implement and validate a script that monitors and gathers the system calls of the spectrum sensor used in this work.
- The creation of a dataset modeling the internal behavior of the ElectroSense sensor while it is operating normally and while it is under attack from the different malware vectors. The dataset contains a subset of datasets each containing the behavior monitored during different malware attacks. Each dataset has as features, the absolute time at which a specific system call was executed, the process that executed it, as well as the process identification number (PID) and the system call name.
- Validation of the dataset through a statistical analysis involving system call frequency and n-gram frequency for n = 2.

1.3 Thesis Outline

The outline of this work is as follows. Chapter 2 provides a the necessary background knowledge to understand the work done in this thesis. Chapter 3 reviews work previously done related to existing datasets and different IDSs. Then, Chapter 4 highlights the system call gathering and monitoring approach proposed in this work. It also depicts the architecture of the monitoring script, as well as the whole environment setup in which the data is collected. Subsequently, chapter 5 presents the scripts developed in order to analyzes and review the gathered results statistically and evaluates the results. Finally, this work ends with Chapter 6 with a summary and conclusion of the work done and a future work outlook.

Chapter 2

Background

This chapter covers all the necessary information needed to understand the work that was done in this thesis. Firstly, it covers device fingerprinting. Secondly, it presents a definition of system calls. Thereafter, it elaborates on different pre processing methods used for system calls. Finally, this chapter closes with expanding on malware affecting IoT devices.

2.1 Device Fingerprinting

The goal of device fingerprinting is to establish a fingerprint, also referred to as a base line, of a devices behavior to be able to compare and understand the behavior of the device when it is operating normally and when it has been compromised [6]. Different data produced by the device such as resource usage, hardware events registered through the hardware performance counters(HPCs), system calls, software signatures, network communication and the data collected through the devices sensors can be used to create a devices are some of the possible ways to create a behavioral fingerprint of a device [19]. In the case of IoT devices, generating these behavioral fingerprints is a challenge, especially because of the multitude of devices produced by multiple manufacturers, each with their own protocols and procedures [6]. System calls allow for the creation of OS-based behavioral device fingerprints [20].

2.2 System Calls

System calls are defined as the interaction between user programs and the kernel of the OS [81]. System calls are the link between user mode and kernel mode. They are used when a process or program needs to hand over information to the kernel or request resources [7]. In a Linux environment, system calls provide the necessary interface for an application to communicate with the Linux Kernel [11].

2.3 System call Pre-Processing

The previous sections discussed behavioral fingerprinting and the serviceability of system calls in creating an accurate behavioral base line for a device. This section explores different system call pre-processing methods, which are used to represent the system calls in a way that not only statistical information can be inferred from them but also, to be able to feed this data to ML or DL based IDSs in order to train them to more accurately and promptly detect zero day attacks. methods that are used for intrusion detection systems. This section focuses on the frequency, sequence and graphs of system calls.

Frequency based approaches rely on the calculating the frequency of occurrences when compared to other system calls. Liao and Vemuri [21], for example, treat all system calls generated by a program as single "words". These words then build a single array of system calls, which they refer to as "document". Thereafter, text analysis models such as the k-Nearest Neighbour (KNN) ML algorithm can be applied to train an IDS [21]. Another frequency based method, used to extract features out of a system call sequence is the N-gram vector. Xu et al. [76] define an N-gram as a finite sequence n of system calls of length n. With the parameter L being the number of unique system calls and n being the the length of a contiguous sequence of system calls, for any given system call trace there exist L^n different n-grams. Together with the support vector machine classifier and various system call graphs created with the n-gram data, 87.3% of classification accuracy was achieved [76]. Furthermore, after thorough investigation, Tan et al. [77] concluded for datasets UNM [22] and ADFA-LD [68] a 6-gram and a 7-gram provide the best performance for those datasets respectively [77] [78]. In conjunction with the N-gram algorithm, the sliding window method with window size n mostly used to iterate over the whole system call trace to then generate n-grams of system calls [81].

In contrast, another set of methods are based on using the sequence of system calls. One of these algorithms is known as sequence time-delay embedding (stide) [40]. This method firstly, removes all parameters from the system calls to reduce complexity and computational resource and secondly, enumerates all unique adjoining sequences of system calls of a predefined length [40]. Stide uses the sliding window algorithm across each trace and adds each unique sequence to a separate database, where they are stored as trees for performance reasons [40]. Pairing stide with a Hidden Markov Model (HMM) classifier delivered 96.9% detection accuracy [40]. Furthermore, Dymshits et al. [38] designed a compact format where sequence of system calls are transformed into time constrained count vectors, which were then used by long short-term memory (LSTM) and recurring neural network (RNN) classifiers to deliver a detection rate between 90% and 93%.

A further method to pre-process system calls is by presenting them as graphs. Mpanti et al. [41] propose a method in which the system calls generated by a single program are portrayed as direct acyclic dependency graphs. The vertices symbolize and the edges symbolize the system calls and their flow respectively. Next, a more abstract version of these graphs is produced by grouping system calls by functionality. Lastly, temporal evolution graphs are created in order to detect and classify anomalous behavior [41]. A further instance of system call graphs is presented in the work by Grimmer et al. [42]. They portray each programs system calls n-grams sequences as a graph with the vertices

signifying a system call and the edges the transition from one system call to another [42]. Thereafter a broader probability graph is derived from all graphs. The frequencies of each transition and also the probability of the transition are calculated and added to the edges. These probability graphs are then feed to classifiers such as nearest neighbor and k-centers [42].

Table 2.1 below, presents an overview of the different pre-processing methods. The methods and works listed are separated through a double horizontal line to signify a change in pre-processing method.

Table 2.1: System call Pre-Processing & Classifiers

Pre-processing Approach	Approach in detail	Classifiers	Devices	Malware Detected	Detection Accuracy
[21] Frequency	Each program produces a "document" of system calls	KNN	-	DoS, Buffer overflow	91.7%
[76] Frequency	N-gram	SVM	Genymotion android emulator	-	87.3%
[40] Sequence	stide, T-stide	HMM	Unix devices	Buffer overflow, Trojan backdoor	96.9%
[38] Sequence	10 second system call count vectors	LSTM & RNN	Linux devices	-	90-93%
[39] Sequence	Create a feature vector using n-gram with n = 1,2,3	Random Forrest (RF), Native Bayes (NB) & SVM	IoT MIPS running Linux	DDoS, Bashlite, Hajime	Up to 97.44%
[41] Graphs	Temporal evolution graphs	Cover & delta similarity metrics	-	-	-
[42] Graphs	Portray n-grams as graphs with probability & frequency edges	KNN & k-centers	-	Java & Linux meterpreter	80%
[37] Graphs	Converting system calls into a system call graph	Convolutional Neural Network (CNN)	QEMU ARM	IoT Botnets	97.22%

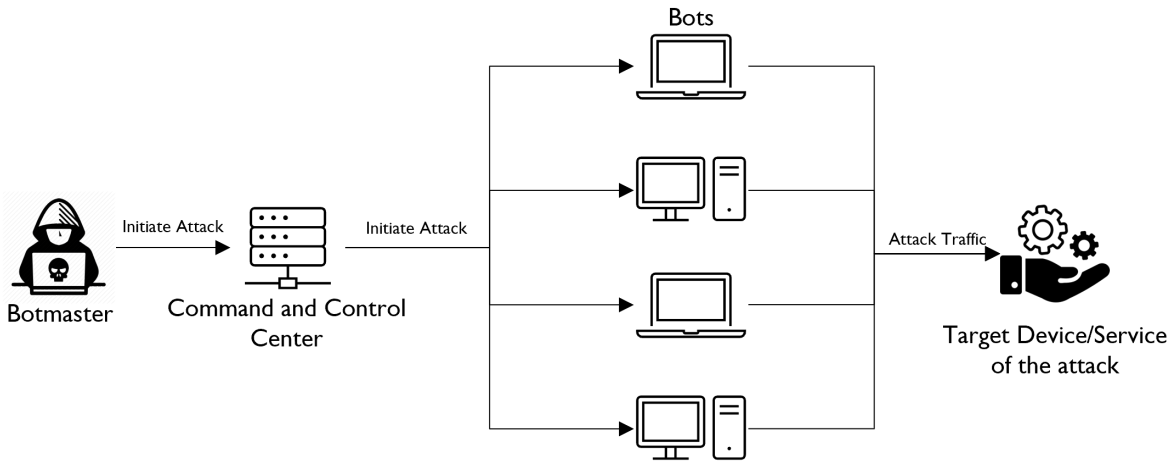
2.4 Malware

Spectrum sensors, due to their resource limitations, are vulnerable to an array of different malware. This work covers Botnets, Backdoors, Ransomware and Rootkits as they are among the most common and harmful cyber-attacks for IoT ElectroSense sensors. Table 2.2 provides an overview of the different malware vectors that exist for the malware families mentioned above. The table is subdivided into malware families which are denoted by two horizontal lines. Also the table lists the behavior and main objective of each malware vector.

2.4.1 Botnet

A botnet is a network of infected devices, also called bots, that can be remotely controlled by an attacker, which is often referred to as botmaster. The infected devices can also be IoT devices. The attacker makes use of the botnet through an interface called Command and Control (C&C) server. Through the C&C server, the botmaster can to carry out attacks such as a distributed denial of service (DDoS) [24]. The C&C server is responsible for broadcasting the attackers commands to the infected devices and maintaining their connections. The anonymity gained through this interface and the fact that the infected bots are distributed across the world are the key aspects of what makes botnets so attractive for cybercriminals. The DDoS attack is one of the most common attacks launched from a botnet. DDoS attacks try to interfere and interrupt either a user's connectivity by exhausting network and transport layers or user services by depleting the server resources [35]. Figure 2.1 illustrates the general infrastructure of such an attack. Other more advanced botnets like for example Mirai [29] offer extra features. For example, the bots in a Mirai botnet not only carry out commands specified from the botmaster through the C&C, but they also actively search for more devices that can be added to the botnet and report back to a report server which then ensures that these new devices are added to the botnet by executing the bot compiled to the corresponding architecture [12]

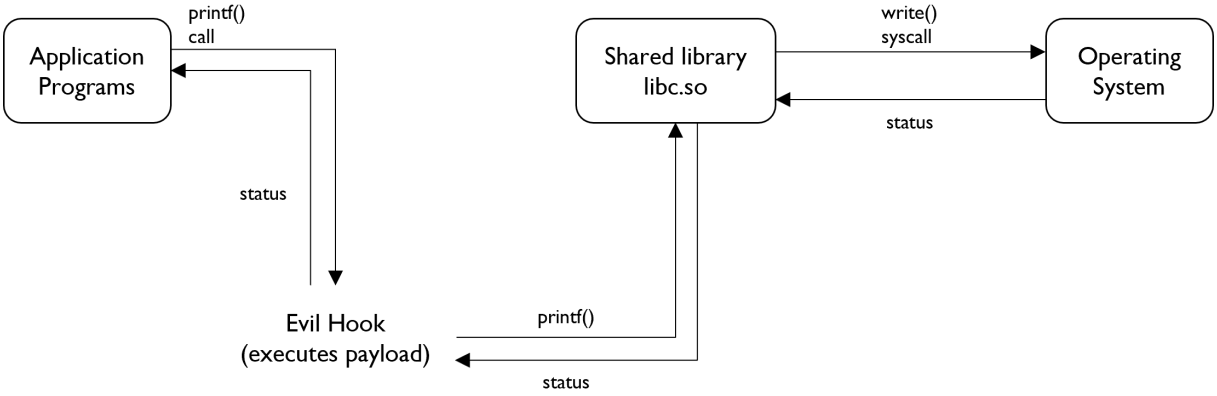
Figure 2.1: Botnet DDoS Attack



2.4.2 Rootkits

Rootkits are a set of software tools that are generally used by an attacker to obtain administrative privileges, also referred to as root privileges, indefinitely and hide their malicious activity on a target device [36]. Besides gaining root privileges of the target system, the main goals of a rootkit is granting unauthorized access to the target device, hiding it's functions and processes, either by cleaning system logs, modifying system commands and the operating system Kernel and ensuring control over the target system even after rebooting [53]. Rootkits are generally classified in two groups. On the one hand, *LD_PRELOAD*, also called "user mode rootkits", take advantage of the LD_PRELOAD environment variable to inject ill natured libraries that overrides the actual libraries and their functionality [17]. On the other hand, the other group is called Loadable Kernel Module (LKM) rootkits. They modify where the pointers are directed to in system calls. They can also abuse device drivers and kernel probes [17]. Rootkits are hard to detect, nonetheless the inherent modification of how the functions behave creates a detectable footprint in form of malicious code either in the user space or kernel space [17]. Moreover, rootkits are often used to create backdoors in the target device. A visual representation of how a user mode rootkit works and how it intercepts a call from a user space application before it reaches the dynamic linker is provided in Figure 2.2 below. This interception allows the rootkit to load its own code to the dynamic linker without the user nor the OS noticing a change.

Figure 2.2: User mode rootkit. Source: Adapted from [18]



2.4.3 Backdoors

Backdoors are a type of malware that allows for continuous legal or illegal access to a device by bypassing system authentication and gaining root privileges [43] [13]. Legal access because often times, backdoors are installed onto IoT devices by the company producing them for debugging and security purposes [14]. These pre-installed backdoors can be exploited by attackers. In addition, backdoors can be installed on a device through rootkits or trojans [13]. Backdoors permit the attacker to then remotely execute commands on the target device, launch a DDoS attack, access and retrieve system information and confidential data. Moreover, backdoors serve as a gateway for other types of malware being installed on the target device [13]. Ransomware and Spyware are among the malware that can be installed through a backdoor. Also, backdoors allow the attacker to delete data stored in the target device, reboot it and also include it in a botnet.

2.4.4 Ransomware

In a ransomware attack, the attacker infects a system and encrypts its data until the user of the target system pays a ransom fee defined by the attacker [15]. There are different sub types of ransomware. On the one hand, crypto ransomware encrypts important data files from the target device. This type of ransomware targets mostly systems where large amounts of data are located [16]. Generally, the attacker demands that the ransom payment is paid in cryptocurrency or other method that is hard to trace [16]. Figure 2.3 depicts the workflow of crytp ransomware. This type or ransomware typically finds its way through spam mail. On the other hand, the locker ransomware prevents the user from accessing the infected machine entirely and also alters the way in which the IoT device behaves in order to coax the victim into paying the ransom [16].

Figure 2.3: Crypto Ransomware attack. Source: Adapted from [15]

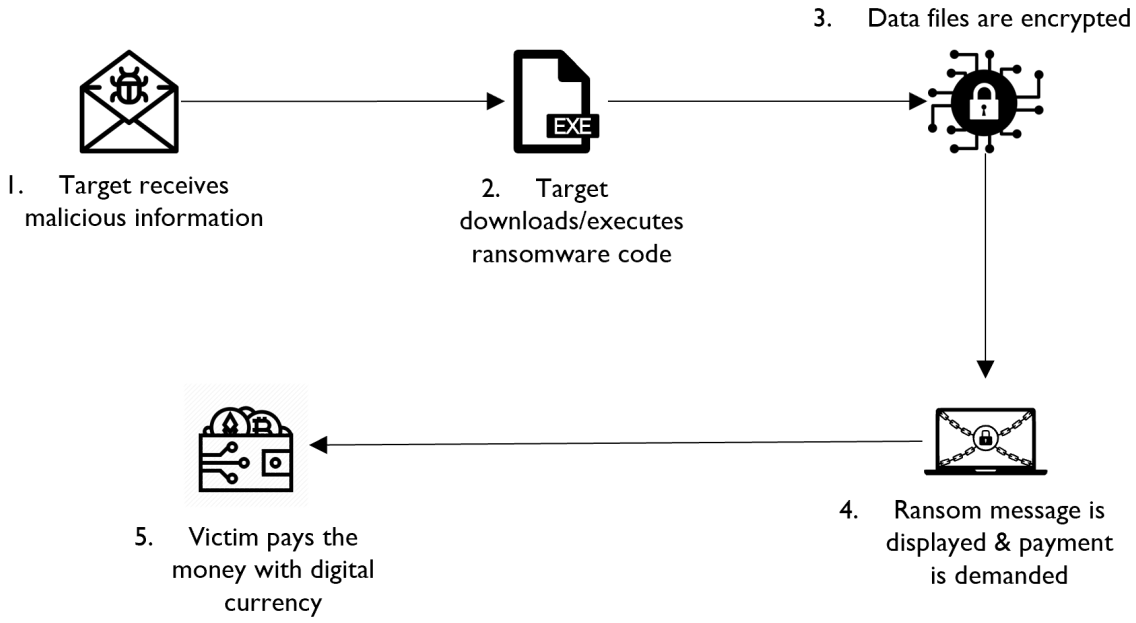


Table 2.2: Malware types and specific vectors

Vector	Malware Family	Behavior	Goals
backdoor[31]	Backdoors	Server sends commands to IP of raspberrypi on given port and extract files from the raspberrypi to the attackers device	Open a shell on the target device
httpBackdoor [30]	Backdoors	Retrieve system information like SSH keys through GET requests and execute command line commands through POST requests.	Gather system information and execute commands
thetick[32]	Backdoors	C&C structure where the attacker can send commands to multiple clients	Open shell on target device
Bashlite[25][24]	Botnet	Grow botnet through brute forcing telnet credentials	Grow botnet for DDoS attack
Mirai[29][24]	Botnet	Allows bots to scan for vulnerable devices. Resolves C&Cs IP addresses using DNS. Communication with binary protocol	DDoS attack which operator may also sell through web interface

Persirai [23]	Botnet	Access webcam interface through TCP port	Tries to get access to the webcam related router through plug and play (UPnP) vulnerability to download and execute the malware binaries
Hajime [23]	Botnet	Similar infection method to Mirai but makes use of BitTorrent DHT protocol for distributed communications	Every message is encrypted and signed using public and private keys.
BrickerBot [23]	Botnet	Uses SSH default credentials and known vulnerabilities to gain access to the device	Attempts a permanent denial of service either through defacing a device firmware, erasing all files and reconfiguring network parameters
Ransomware-PoC [34]	Ransomware	Generates AES key which is encrypted by the attackers RSA public key	Encrypts target device directories with AES key
EKANS[33]	Ransomware	Scan domain for controllers to compromise	Encrypt files and display ransom note
Bdvl[27]	Rootkit	Preload	Hidden backdoors, keylogging and gaining access to passwords and files
Diamorphine [28]	Rootkit	LKM	Hide high CPU usage by hooking read() and sysinfo() syscalls. Also has the ability to hide/unhide any process
Beurk[26]	Rootkit	Preload	Hidding backdoor clients, files and directories. Also real time log clean-up

Chapter 3

Related Work

This chapter discusses the literature related to this thesis. Firstly, an overview of the existing behavioral datasets that have been created over the years to train different Intrusion Detection Systems (IDSs) is provided. Secondly, an overview of different types of IDSs is presented.

3.1 Datasets modeling device behaviour

With the rise of IoT usage it has now become more important than ever to understand the internal behavior of these devices when they are under attack and control of different types of malware to be able to take action accordingly. This is where datasets and IDSs come into play. This section discusses an array of datasets, both network and host based, used to train different IDS. First it reviews multiple network based datasets and then it transitions to explore host based datasets.

3.1.1 Network Datasets

Network based datasets focus on storing the network generated data of the intercommunication between devices in a network. They contain mostly transfer protocol data generated by protocols such as HTTPS, HTTP, FTP and others.

The Center of Applied Internet Data Analysis (CAIDA) organization has created multiple datasets from 2002 to 2016. On of these datasets is the CAIDA DDOS Attack from 2007, which contains almost an hour of DDoS attack generated traffic traces which have been anonymized [55]. One of the limitations of this dataset is that normal non-attack traffic is not present, making the dataset an inaccurate and asymmetric representation of real world network flows [55] [56].

Produced over a seven-day period in 2012 by the Information Security Center of Excellence (ISCX) from the University of New Brunswick (UNB) the ISCXIDS2012 is a dataset that

contains regular and irregular network traffic [63]. The dataset is comprised of multiple alpha and beta profiles. The alpha-profiles contain descriptions of attack scenarios and the beta-profiles describe distributions and behaviors of different entities such as packet sizes of different protocols [57]. These profiles are then used to generate attacks and realistic network traffic. Attack scenarios like HTTP denial of service, IRC (Internet Relay Chat) DDoS and brute force SSH and network protocols like HTTP, IMAP and SMTP can be found in the beta-profiles [52, 57]. Drawbacks of the ISCXIDS2012 dataset are the limited traffic protocols and the absence of HTTPS, which accounts to nearly 70% of current network traffic [63]. To make up for these limitations the Canadian Institute for Cybersecurity dataset (CICIDS2017) was released. The beta-profile for this dataset is comprised of not only the protocols contained in ISCXIDS2012 but also the broadly used HTTPS network protocol [52]. The Alpha-profiles are composed of brute force, heart-bleed, botnet, DDoS, web and infiltration attacks targeted at different Windows and Ubuntu based operating system servers and computers [52]. Shortcomings of this dataset include high class imbalance, which if used for training of a classifier, induces lower accuracy and higher false alarm rates [62]. Also as per Panigrahi et al. [62] the CICIDS2017 dataset has "[...] 288602 instances having missing class label and 203 instances having missing information.".

Moreover the dataset CSE-CIC-IDS2018, a collaborative effort between the Communications Security Establishment (CSE) and the CIC, is an update to the CICIDS2017 dataset that is much larger and contains over 16 Million instances of simulated network traffic [66]. Nevertheless, both the CSE-CIC-IDS2018 and CICIDS2017 datasets have drawbacks, such as high class imbalance which results in low detection accuracy and a high false positive rate [66].

The KDD Cup 1999 dataset (KDD) [50], created by Stolfo et al. [49], is based on 7 weeks of network traffic raw tcpdump data provided by the 1998 DARPA Intrusion Detection Evaluation Program [64]. The dataset contains traces of simulated attacks of the likes of DoS, unauthorized remote access, user-to-root privilege escalation and probing attacks [49]. Tavallaee et. al. [48] investigated the shortcomings of this dataset and found that, apart from it all being synthetic data that does not resemble the real world, due to the large amounts of redundancy present in the dataset causes learning algorithms to be biased, in a way that it decreases the probability of discovering harmful attacks over the network. Another inherent problem of the KDD dataset is that the network attack taxonomies are not well defined, there is no concrete threshold of what is classified as an attack and what not [48].

To solve some of the issues of the KDD the Network Security Laboratory (NSL) KDD dataset was created. The NSL KDD dataset does not contain redundancy in the training set, causing the classifiers to not be biased towards more frequent traces [51]. Also, there is no duplication of records in the test set, consequently bettering the performance of the learners [51].

The Defense Advanced Research Agency (DARPA) commissioned Linconln Laboratory 1999 dataset is an artificially created dataset used to benchmark intrusion detection systems [45]. The dataset was designed to resemble both the network traffic and host audit logs generated by an US Air Force base [44]. The network traffic was generated through

automata-based programming that simulated network behavior different users such as secretaries, system administrators, programmers, attackers and other users [46]. It contains three weeks of training and two weeks of test data with 58 attack types, all executed against UNIX and Windows NT hosts [44]. The dataset contains different DoS attacks, multiple user-to-root attacks and root-to-user attacks [46]. The major drawback of this dataset, as Brown et al. [47] point out, is that the real life network behavior differs from the one portrayed in the dataset. Also, because of the rapid evolution of network attacks this dataset has become outdated, thus training an IDS, more specifically a NIDS, with this dataset would bring very low detection rates [54].

3.1.2 Host Datasets

Host based datasets on the other hand, are datasets that store system internal activities. Instead of focusing on the communication between devices, host based datasets focus on the internal behaviour. Data such as system calls and logs are commonly stored in these datasets.

The Australian Defence Force Academy Linux Dataset (ADFA-LD) dataset was developed to address the shortcomings of the outdated KDD datasets [68]. The malware featured in this dataset create a more realistic depiction of a system being completely under attack, from the infection phase all the way to the execution phase [67]. The dataset includes attack traces from attacks like web exploitation, poisoned executables, remote password brute-force and trigered vulnerabilities, social engineering and system manipulation using webshell [67]. According to Creech et al. [67] the dataset is made up of 833 normal system call traces, 4373 normal traces to analyze false alarm rates and 6 different types of attacks spread out in 746 traces for testing [81]

Another dataset also produced by the Australian Defence Force Academy is the Windows Dataset, ADFA-WD, the Windows OS counterpart to the ADFA-LD. It consists of a collection of dynamic link libraries (DLL) access requests and system calls [69]. The target system used was based on a Windows XP Service Pack 2 distribution with a running FTP and web server, a management tool, a streaming audio digital radio package, and a wireless and ethernet networking connectivity with a fake wireless access point [70]. The nine DLL calls were selected based on it being able to represent system behavior, its capability of representing modern threat vectors and its effectiveness in reaching efficient training and testing in the Host based Intrusion Detection System (HIDS) decision engine [70]. Furthermore, 12 known vulnerabilities to the target device were made use of. These vulnerabilities allow for exploits such as metasploit attacks, reverse ordinal payload injection, blind shell spawning to name a few [69]. The dataset contains 356 traces of normal training data, 1828 traces of normal validation data and 5773 attack traces [70].

Furthermore, the dataset ADFA-WD:Stealth Attacks Addendum (ADFA-WD:SAA) is an extension of the ADFA-WD dataset. It contains 863 new attack traces, fewer compared to the ADFA-WDs 5773 attack traces. It is expected to be used in conjunction with ADFA-WD. The focus of ADFA-WD:SAA is mainly on using stealth attacks for generating traces of DLL calls. The three stealth attacks used were Doppelganger, Chimera and Chameleon. The Doppelganger attack exploits the target device by using normal system

functions. Firstly, it adds a new user with elevated privileges on the target device. Then, a new shellcode creates a remote access instance, granting access to a remote third party. Moreover, the Chimera stealth attack exploits a valid process running on the target system and spawns a new process which is then used to connect back to the attacker. The third and final stealth attack of the ADFA-WD:SAA dataset is called Chameleon. It uses an independent excerpt of machine code that creates a command shell without having to rely on system services [67].

In addition to the aforementioned datasets, the University of New Mexico (UNM) datasets contain system call traces of multiple processes [79]. One of the datasets is the *sendmail* dataset. System calls of this program were traced on a device running a Unix based OS called SunOS version 4.1.1 [83]. Furthermore, another UNM dataset contains system call traces of *login* and *ps* processes that were gathered from a single target device running on the Linux Kernel version 2.0.35 [84]. This dataset also contains traces of a Trojan backdoor attack targeted at *login* and *ps* commands [84]. In addition, the dataset containing traces of the Xlock program where it is being interfered with by a buffer overflow attack [85] [82]. Although the UNM datasets have been and are still used to benchmark IDSs, they are outdated and do not contain traces of attacks of current importance [82].

The Firefox dataset [82] was created to address the some of the issues found in the UNM datasets. It consists of both normal and anomalous system call traces. On the one hand, normal behavior system call traces were collected by monitoring standard behavior of the functionality of the Firefox web browser. On the other hand, to gather anomalous traces, several different attacks were launched against the browser. Attacks such as, memory corruption exploit that attempts to execute randomly selected code, integer overflow attack and the exploit of dangling pointers to originate a DoS and execute arbitrary code and Document Object Model (DOM) exploit causing memory corruption are used to generate anomalous behavior [82] [65]. A natural down sight to this dataset is that the focus lies solely on the Firefox web browser.

3.2 Intrusion Detection Systems

An IDS is a threat monitoring systems that can be differentiated into host based intrusion detection system (HIDS) or network based intrusion detection system (NIDS) based on what type of data they work with. NIDSs examine network traffic and network generated data, where as HIDSs analyze system specific internal data and activities and thus has a greater prospect of detecting malware attacks that target the internal workings of a device [65]. Also, in comparison to NIDSs, HIDSs are able to single out which processes are involved in any given attack and at the same time, are able to follow the attacks behavior all the way to its outcome [75].

HIDSs can be subdivided even further to specify what type of data it is that they are analyzing. System-call based HIDSs, for example, dissect, as the name already implies, system call traces [65]. HIDSs, more specifically, system call based HIDSs are an effective approach, however, these systems are currently struggling to handle the ever growing data volume of system call traces that are being generated. This renders them incapable

of executing in depth system call analysis, which are vital for the detection of attacks, especially those that are being executed through multiple hosts [65]. Moreover, HIDSs greatly consume computing resources of the device that it is running on in order to better train the decision engines and in turn better the malware detection rate. This makes current HIDSs software an unfavorable option for IoT devices, like the ElectroSense sensor discussed in this work, as these are low resourced constrained devices [65].

IDSs use datasets to train their decision making engines and also to benchmark their detection rates. Here we differentiate between the datasets targeted for HIDSs and NIDSs usages. HIDSs datasets, as mentioned above, contain internal system behavior data like system calls. By contrast, NIDSs datasets store information of the communication and network traffic between devices.

In general it can be observed that the trend of intrusion detection datasets up until now has focused more on the network aspect, which is good to detect only a subset of malware such as botnets, but not other malware like rootkits, backdoors and ransomware. More importantly, most datasets used in IDS currently do not focus on IoT devices and even less on their internal behavior when infected by malware. Also, the malware traces in the datasets that have been used regularly to train IDSs such as, ADFA-LD, UNM and also network datasets such as DARPA and CSE-CIC-IDS2018 have either outdated malware attack traces or false positive rates which are to high to deliver acceptable detection accuracy percentages. This thesis seeks to cover the gaps of the limitations of the current literature, by creating a dataset that models the internal behavior of IoT devices through system calls. The focus here lies especially in gathering the behavior of the device while it is under attack of novel malware of the likes of botnets, backdoors, rootkits and ransomware.

Table 3.1 provides an overview of an array of different datasets that have been created, categorizing them into either internal behaviour or network oriented behaviour datasets. The separation of these two types is signaled through a double horizontal line in the table. Along with the types, the contents, malware attack traces found in the dataset and the drawbacks of each dataset are summarized in the table.

Table 3.1: Existing Datasets

Type	Dataset	Contents	Malware attacks	Drawbacks
Internal	ADFA-LD(2013) [52]	System call traces of various Linux devices	FTP and SSH password brute force, Java Meterpreter and Linux Meterpreter	Lack of attack diversity and attacks in dataset are not well separated from normal behavior
Internal	ADFA-WD(2013) [52]	DLL traces	Metasploit exploit, DNS spoofing, reverse shell spawn through PDFs	-

Internal	ADFA-WD: SAA (2013)	DLL traces of three stealth malware	Doppelganger, Chimera, Chameleon	-
Internal	UNM (1999) [52]	System call from various processes	Trojan backdoor, buffer overflow and DoS	No system call arguments and not representative of today's attack diversity
Internal	Firefox DS [82]	Normal and malicious system call traces generated by Firefox web browser	Memory corruption exploit and DoS	Focus lies on Firefox web browser only and it has not been used in many case studies
Internal and Network	NGIDS-DS [71]	Host based logs and network packets	DoS, Shellcode and Backdoors	Only used in few studies
Network	DARPA (1998-99) [45]	FTP, Telnet and e-mail activities	DoS, Buffer overflow, remote FTP	Outdated infrastructure and attack types
Network	CSE-CIC-IDS2018 [59]	HTTPS, HTTP, SMTP, POP3, IMAP, SSH	DoS, SSH Bruteforce, DDoS, Botnet	Class imbalance
Network	CIC-IDS-2017 [58]	Network traffic for HTTP, SSH and IMAP protocols with payload	Brute Force FTP, Brute Force SSH, DoS, Heartbleed, Web Attack, Infiltration, Botnet and DDoS	High false positive rate because of class imbalance
Network	ISCX2012 [60]	HTTP, SSH, SMTP, POP3, FTP and IMAP network traces	DoS, DDoS, Brute-force SSH	Outdated as HTTPS protocol not present
Network	CAIDA DDoS (2007) [55]	DDoS attack traces split into pcap files	DDoS	Very specific to a particular attack
Network	NSL KDD (2009)	Raw tcpdump packets	DoS, Remote to Local, Probing and User to Root attacks	Synthetic data
Network	KDD Cup (1999)	Raw tcpdump packets	DoS, Remote to Local, Probing and User to Root attacks	Synthetic data, no exact definition of network attack, record redundancy
Network	Kyoto (2009) [61]	DNS and mail traffic	-	Synthetic data and no false positives

Chapter 4

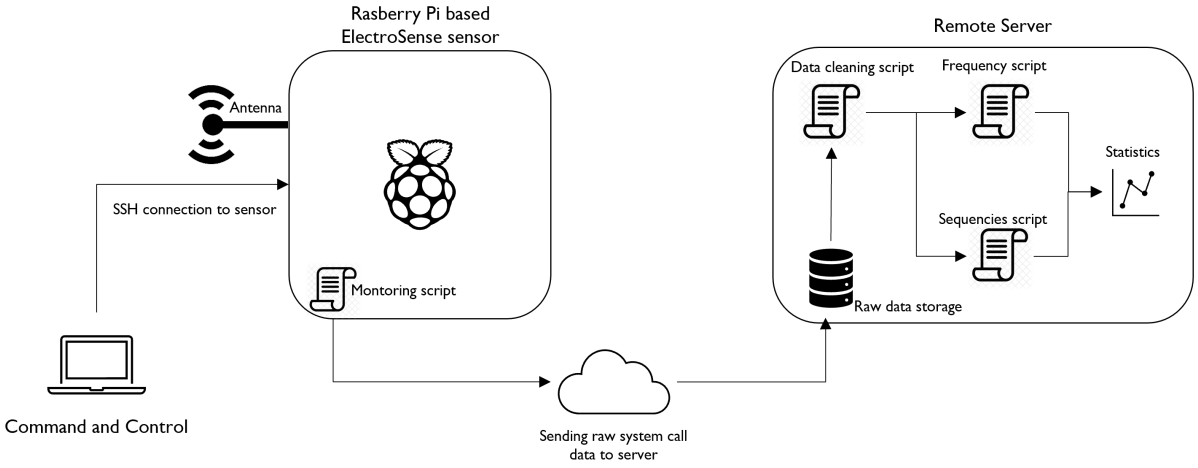
Creation of a System Call based Dataset

4.1 ElectroSense Scenario

In this work we look at a spectrum sensor, more specifically an ElectroSense sensor as our target device. The ElectroSense sensor is part of the ElectroSense network. ElectroSense is a crowdsensing platform that senses the radio frequency spectrum. Multiple Raspberry Pis with an sdr frontend and an antenna act as the sensors that gather the spectrum data. This data then gets sent to the ElectroSense backend. The backend system follows a three layered architecture, also known as Lambda architecture, with a batch, a speed and a serving layer which allows for greater system scalability if needed [74]. Data sent to the backend from the sensors is then processed and the results are made available to the users via open Application Programming Interface (API) [74]. The Raspberry Pi acting as the embedded sensor used in this work is a Raspberry Pi 4 with an ARMv7 rev 3 processor and 4 GB of RAM. Attached to the Raspberry Pi is an antenna that senses and collects the radio frequencies that are then processed by the sdr radio frontend and then sent to the ElectroSense servers. The sensor can sense frequencies between the ranges 20 MHz and 6 GHz. For this work, the ElectroSense sensor is connected to the internet via LAN cable and has a static IP address.

In order to create datasets modeling the internal behavior of these devices, this work focuses on gathering system calls from the device and creating datasets out of them. To capture the system calls from the sensor, a custom script is running on the device which gathers the system calls in a text file format and proceeds to send them to a remote server. The remote server in turn, has a script running that cleans the raw data and stores the files in a by attack, date and hour folder structure. Additionally, in order to evaluate the datasets and calculate the mean and standard deviation of the system call frequencies and sequences on a per malware basis, two scripts are executed on the cleaned data. Figure 4.1 describes the setup.

Figure 4.1: System Call Monitoring and Analytics Process High Level



4.2 System Call Monitoring Process

Different Linux commands to gather the ElectroSense system calls were reviewed. Among those, perf trace [72] and `strace` [73] stood out as good candidates. In the end `perf trace` was chosen for this work, as it gathers the system calls of the whole system, unlike `strace`, that only gathers system calls of user specified processes. Gathering system calls for only a handful of pre-execution specified processes would limit our ability to begin monitoring new malware-induced processes and thus render the system call gathering useless.

4.2.1 Architecture

The system call monitoring architecture consists of 2 scripts. The first one *monitoringScript.sh* runs on the ElectroSense sensor itself. It consists of the following code blocks:

- Setup
  Here, the user can specify variables such as duration of the script execution in seconds, minutes or days, how long the system call intervals should be in seconds, after how many iterations the files should be transferred and the `RESULTS_PATH` in which the intermediate results should be stored. Furthermore, the user must specify an IP address and folder directory where the gathered data should be transferred, listed in the pseudo code below as `RSYNC_PATH`. If the duration of the script or the interval duration are not set, the default values, 2 days and 10 seconds will be used respectively. However, if one or both `RSYNC_PATH` or `RESULTS_PATH` are not specified by the user, the script will not work. The first section of the pseudo-code below in Algorithm 1 visualizes the flow of the setup.

Algorithm 1 Monitoring Script Pseudo-Code

```
/*Setup*/
Require: RSYNC_PATH ≠ NULL
Require: RESULTS_PATH ≠ NULL
Require: TIME
Require: SLEEP

if TIME = NULL then
    TIME ← "2days"
end if
if SLEEP = NULL then
    SLEEP ← 10
end if
if ITER_MAX = NULL then
    ITER_MAX ← 1
end if

/*Main Loop*/
while CURRENT_TIME ≤ START_TIME + TIME do

    /*Data Transfer Section*/
    if ITERATIONS = ITER_MAX then
        ITERATIONS = 0
        for FILE in RESULTS do
            rsync FILE to RSYNC_PATH
        end for
    end if

    /*Data Gathering + Output*/
    CURRENT_TIME = NOW
    perf trace -o /RESULTS/NOW.log -a – sleep SLEEP
    Add EPOCH and UPTIME to NOW.log
    ITERATIONS+ = 1
end while
```

- Monitoring Loop
  This loop, also depicted in Algorithm 1 runs for the duration that was specified in the setup block. It is responsible for the data gathering, output and transfer. Also, the execution of the collection, output and transfer blocks happens every certain amount of time according to what was set in the setup block.

- Data Gathering and Output
  This is the key component of the monitoring script. The `perf trace` command is called and it runs for a predetermined amount of time set in the setup section. When its done, the output gets written to a temporary file that later on is transferred to another, with more storage capacity, computer through `rsync` as shown in the Data Gathering + Output section in the Algorithm 1.

- Data Transfer
  After the `perf trace` command has reached the desired iterations, all the files temporary files are then transferred to another computer with more storage capacity via `rsync` and the iterations counter `ITERATIONS` is set back to zero, as mirrored in the Algorithm 1.

As it is important to know if the *monitoringScript.sh* script interferes with the normal operation of the sensor, while active. For this reason, the *monitoringScript.sh* script was also monitored for resource consumption using the `top` Linux command. On average, the `perf trace` command used in the script consumed 26.65% of one CPU resource while the `rsync` command consumed 74.1% of one CPU resource. As the ElectroSense sensor has 4 CPU cores, the above mentioned percentages can be divided by four, as each core has its own 100% capacity. So the real CPU consumption values are 6.6625% and 18.525% respectively. Also, the average memory usage of `perf trace` is 0.2% and for `rsync` it is 0.1%. Both the CPU and memory usage of the *monitoringScript.sh* script are low and do not interfere with the normal behavior of the ElectroSense sensor.

The second script, *data_cleaning.py*, focuses on cleaning the raw data so that it can be pre-processed more effectively to extract critical information, such as statistical data. The script removes all arguments from the system calls. Ideally it runs on the machine were the data is being sent to from the *monitoringScript.sh* script. A pseudo-code version of it is depicted in Algorithm 2 and it is organized as follows:

- Setup
  In this section, the user must specify three directory paths, one from where the script should retrieve the raw system call data, another for where to store the processed data and a third one to specify the location where the system call per process summary provided by `perf trace` should be saved. Also, the user must specify for how long the script should run, this as an int specifying an epoch time in the future. If any of these variables are not set, the script does not work.

- Main Loop
  The program loop runs for a variable amount of time, also specified in the setup section. This loop is responsible, as the name suggests, for the whole script. It contains the secondary data extraction section, as well as the data cleaning and data outputting sections

- Secondary data extraction
  This part of the code manages the extraction of the `EPOCH` and `UPTIME` of each system call. These values are later on used to compute the absolute time at which each system call was executed. Also, this section is in charge of transferring all the summary data provided by the `perf trace` command onto a separate file, named exactly as the original file but with a *-summary* tag attached to it. These summary files are saved onto a directory, specified by the user in the setup section.

- Data cleaning
  Furthermore, the data cleaning section removes some data from the raw system call input files such as system call parameters. It calculates the time each system

call was called by combining the `EPOCH` time and the ElectroSense sensors `UPTIME`, already extracted in the secondary data extraction section.

- Data Outputting
  The data outputting section saves the cleaned system call data files onto a directory, with sub directories ordered by date and hour time.

Algorithm 2 Data Cleaning Script Pseudo-Code

```
/*Setup*/
Require: destination_dir ≠ NULL
Require: input_dir ≠ NULL

if time = NULL then
    time ← "2days"
end if
/*Main Loop*/
while current_time ≤ start_time + time do

    /*Secondary Data Extraction*/
    for file in input_dir do
        if file ends with .log then
            EPOCH = file.get(EPOCH)
            UPTIME = file.get(UPTIME)
            writefile.get(Summary)tofile_summary.txt
        end if

        /*Data Cleaning*/
        for line in file do
            abs_time = line.get(abs_time)
            process_name = line.get(process_name)
            pid = line.get(pid)
            syscall_name = line.get(syscall_name)
            write line to output.log abs_time, process_name, pid, syscall_name
            truncate output.log
        end for

        /*Data Outputting*/
        if destination_dir + fileHour exists then
            save output.log to destination_dir + fileHour
        else
            create destination_dir + fileHour folder
            save output.log to destination_dir + fileHour

        end if
    end for
end while
```

4.3 Malware affecting ElectroSense

The following section elaborates on the four different malware families specific implementations chosen in this work and how they were executed.

4.3.1 Bashlite

Bashlite [25], also known as Gafgyt, LizardStresser, Lizkebab, Torlus and Qbot is a botnet designed to infect IoT devices and add them to a botnet to carry out DDoS attacks. In order to execute this botnet a few changes have to be made to the source code [25]. Mainly, in the client.c file the IP address of the C&C server has to be specified like shown in figure 4.3. Also, as shown in figure 4.2 in the server.c file the management port and password can be changed as desired. The setup of this botnet for this thesis consisted of a C&C center hosted on a Linux Ubuntu server with IP address and port 192.168.1.10:8888. Next, the bot, in this case the ElectroSense sensor was connected to the C&C center through 192.168.1.10:6667. And finally, a different Raspberry Pi with the IP address 192.168.1.17 was the target of the attacks.

```
#define MY_MGM_PASS "password"
#define MY_MGM_PORT 8888

#define MAXFDS 1000000 // No way we actually reach this amount. Ever.
```

Figure 4.2: Bashlite change management port & password

```
unsigned char *commServer[] =
{
        "192.168.1.10" //This is the IP of the command server (you'll need to change this)
};
```

Figure 4.3: Bashlite set IP address to C&C

Once the bot has connected to the C&C, Bashlite offers the following administrative commands:

- *! PING* checks connection to bot.
- *! GETLOCALIP* gets IP address of the bot.
- *! SCANNER ON/OFF* turns function of bots being able to scan for other bots on or off.
- *! KILLATTK* stops the ongoing attack.
- *! LOLNOGTFO* disconnects from the bot.

Further, it offers the following flooding attack commands:

- *! HOLD <target ip address> <port (0 for random)> <time (in seconds)>.*
- *! JUNK <target ip address> <port (0 for random)> <time (in seconds)>.*
- *! TCP <target ip address> <port (0 for random)> <time (in seconds)> <netmask> <flags (syn, ack, psh, rst, fin, all)> <packet size> <time poll interval (in seconds)>*
- *! UDP <target ip address> <port (0 for random)> <time (in seconds)> <netmask> <packet size> <time poll interval>.*

For this thesis, in order to automate the attack procedures for an extended amount of time, the process automation software UiPath [86] was used. The process automation script for Bashlite looped on a bihourly basis through a *HOLD*, *TCP* and *UDP* attack for 30 hours.

A visual representation of the attack execution is displayed in figure 4.4. On the right side console, the C&C center where different attack commands have been entered can be appreciated. Likewise, the left side console illustrates the bot receiving the commands.

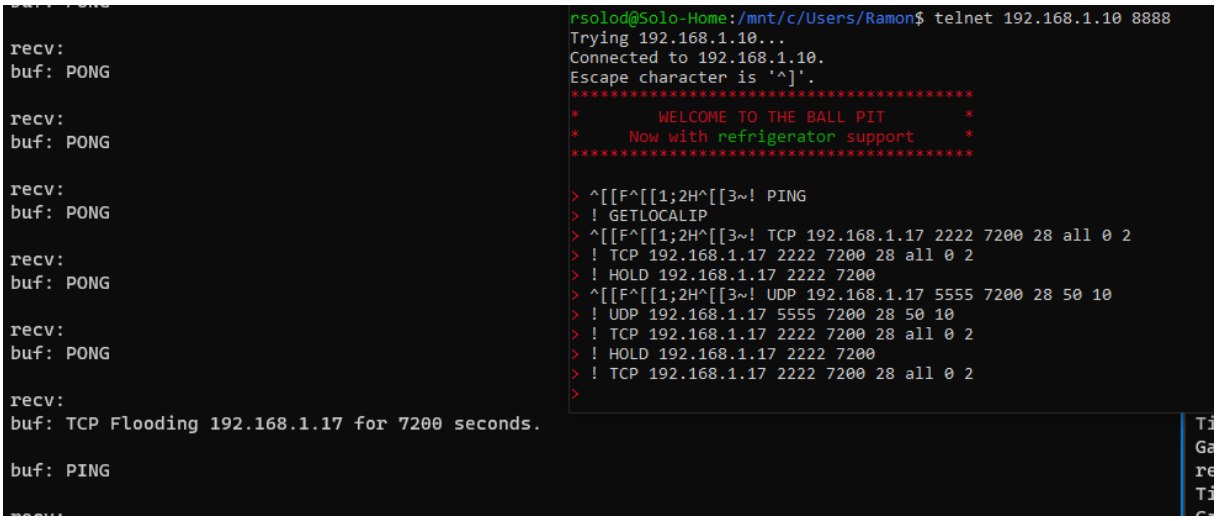


Figure 4.4: Bashlite commands

4.3.2 Bdvl

Bdvl [27] is a LD_PRELOAD rootkit, meaning it runs in the user space of a device. After installation on the target device, it will hide itself from the process memory map files. It supports multiple functionality like port hiding, credential logging from SSH incoming SSH connections. Also it features file stealing, where the attacker can specify what types of file formats and size it should steal. It also offers multiple backdoors. The PAM backdoor is installed after the malware is executed on the target. It creates a backdoor user through which the attacker can log into the remote target. Every directory or file created with this newly created backdoor user

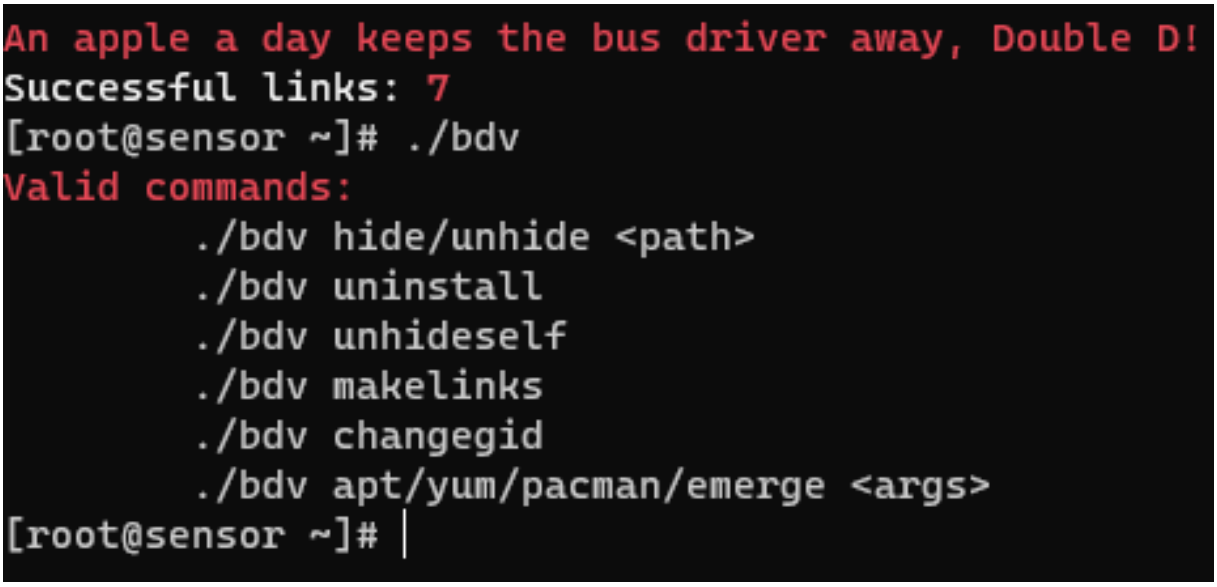


Figure 4.5: Bdvl commands

will be hidden from all other users on the device. Some of the malware's commands include

- *./bdv <hide|unhide> <path>*
  Hides the path of a specific directory.
- *./bdv uninstall*
  Uninstalls the malware from the target device and removes all data that was gathered.
- *./bdv unhideself*
  Unhides the directory where bdvl is installed on.
- *./bdv changeid*
  Changes the rootkits group identifier (GID).
- *./bdv makelinks*
  Creates links that point to all directories of the target device.

Again, UiPath was used to automate the attackers commands onto the target device. After the malware was executed in the remote device, the UiPath script logged into the target device with the new, malware conceived, user. After successful login the malware started stealing data automatically. The scripts loop created a directory, unhid it, hid it again and then proceeded to remove it.

4.3.3 Thetick

Thetick [32] is a backdoor malware designed to affect embedded systems. It has a C&C console, from where the attacker can execute multiple commands on the target device. This backdoor also allows for multiple target devices to be connected to the C&C console, from which the attacker can switch between devices. The attacker can use the following commands, also listed in 4.6

- *bots*
  Returns a list of the clients with their respective IP address and a Universal Unique Identifier (UUID).
- *chmod <file>*
  Changes a file's access mode, much like the existing Linux command.
- *clear*
  Clears the screen.
- *current*
  Shows information on the currently selected bot.
- *dig <domain name>*
  Resolves local domains at the target network.
- *download <url> <remote file>*
  Downloads a file to the target bot via HTTP.
- *exec <command>*
  Executes a non interactive command.
- *exit*
  Exits the console.
- *fork*
  Creates a new bot instance that connects automatically.
- *help*
  - Shows a list of available commands.
  *help **
  - Shows help for all commands.
  help <command>
  - Shows help for selected command.
- *kill*
  Terminates the connection to the currently selected bot.
- *pivot <listen on port> <connect to IP address> <connect to port>*
  Creates a TCP tunnel.
- *proxy [ls]*
  - Lists all active proxies.
  proxy [add] <port> [bind address] [username] [password]
  - Adds a new proxy.
  proxy rm <port>
  - Removes an active proxy.
- *pull <remote file> <local file>*, Copies a file from the target device.
- *push <local file> <remote file>*, Copies a file to the target machine.
- *rm <remote file>*, Deletes a file on target device.
- *shell*, Initializes an interactive shell through the C&C connection.
- *use <>, <IP Address>, <number>, <UUID>*, Selects a bot from the list by UUID, IP address or number. If no argument is passed then the current bot gets deselected.

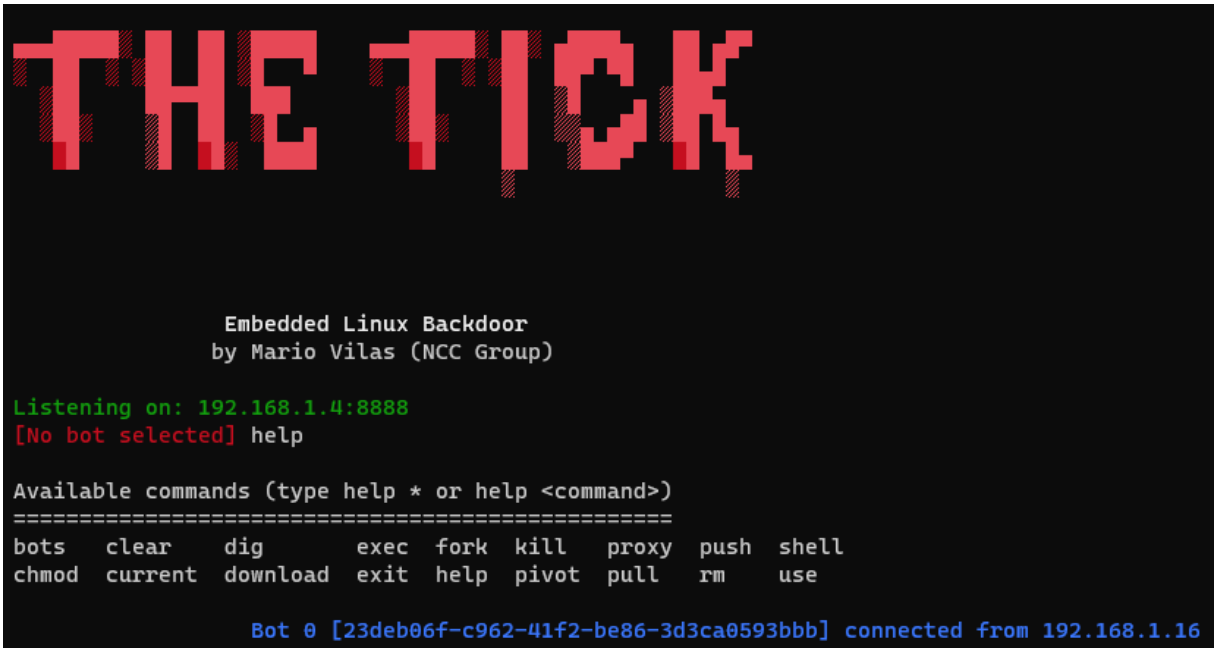


Figure 4.6: TheTick Command & Control console

Like with the other malware instances, a UiPath script was created to automate the actual execution of the malware while the system calls were being gathered. Once script selected a bot then the loop began. It wrote some random text into a file, next, the script copied the newly created file from the target device to the attacking device and simultaneously removed it from the target device. Subsequently, a text file was pushed from the attacking device to the target bot, where it was then also deleted after successful transfer.

4.3.4 RansomwarePoC

This ransomware is designed to encrypt local directories and files with an AES key, which is in turn encrypted with the RSA public key of the attacker [34]. This ransomware has two versions. One, is the basic version which works in the command line only. The second one, extracts the key and sends it back to a CC specified in the source code. Also, a ransom pop up note is displayed on the victims screen. For the purpose of this thesis and because ElectroSense sensors usually accessed through SSH, version 1 was used. The following commands are available for the ransomware.

- *-e* Encrypts the target directory.
- *-d* Decrypts the target directory.
- *-p <directory | file>* Specify which directory or file to encrypt.

Figure 4.7 depicts how the encryption command is executed on the command line. Again, with the help of UiPath, a script that contains a loop which in every iteration encrypted a folder on the target device was created.

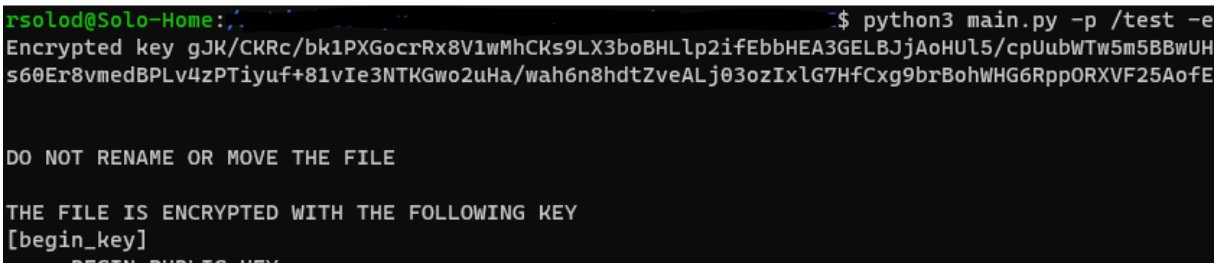


Figure 4.7: RansomwarePoC Encryption

4.4 Datasets creation

As mentioned before, datasets that represent the internal behavior of IoT devices are scarce and this is an issue as datasets are a key component in being able to detect malware in IoT devices, especially zero day attacks with the help of ML and DL combined with a quality dataset. This section elaborates on how multiple system call based datasets were generated, using the Raspberry Pi based ElectroSense sensor.

System calls are a prominent way to model a devices internal behavior. The datasets presented here were constructed with the *monitoringScript.sh* script detailed in subsection 4.2.1. Firstly, the normal behavior of the ElectroSense sensor was monitored by gathering system calls for 48 hours. The normal behavior of the sensor is described as sensing the spectrum data with help of the antenna and sending the gathered data to the ElectroSense backend. Secondly, the *monitoringScript.sh* was executed in the sensor shortly before the device was infected with theTick backdoor. Backdoor commands were executed onto the device for 30 hours in three runs, one lasting 11 hours, the second one 14 hours and the last one 5 hours. The *monitoringScript.sh* was executed shortly before in order to capture the initial infection process in the system calls. Next, theTick was uninstalled from the device and it was made sure that no more traces of the backdoor were found on the device by checking and killing any processes related to it. Then, the LD_Preload rootkit bdvl was installed onto the device and attacks like stealing files, creating, hiding and unhiding directories were executed. Again, the *monitoringScript.sh* script was started before the infection phase. System calls were gathered in one run of 8 hours, two runs of 9 hours and one run of 2 hours, for a total of 28 hours. Subsequently, the rootkit was uninstalled before starting the monitoring script again and infecting the device with Bashlite. The ElectroSense sensor acted as a bot in a DoS attack, sending HOLD, UDP and TCP SYN and ACK attacks to a specified device. The sensor was monitored over the course of a single run of 30 hours with the attacks rotating every two hours. Finally, RansomwarePoC was executed on the device with the help of an automated script created with UiPath [86]. The encryption of a directory was enacted every 15 seconds over the course of a single 30 hour run.

Table 4.1 provides an overview of the created datasets. Each dataset carries the name of the malware with which the device was infected at the time of the system call gathering. It is ordered by malware type and attacks carried out.

Having gathered the raw system calls of the device, the script *data_cleaning.py*

described earlier, was run on all the generated data. This script outputs files with four columns; absolute time, process, process ID and name of the system call without its arguments. In summary, each dataset contains the features absolute time, process name, process id and system call name, with each datasets size ranging from 10 gigabytes and 390 gigabytes.

Table 4.1: Datasets with their corresponding behavior

Dataset	Malware Family	Time(hours)	Attacks/Behavior
Normal	-	48	Normal ElectroSense
Thetick	Backdoor	30	Backdoor, hidden shell, exec commands
Bdvl	Rootkit	28	Stealing files, hiding and unhiding directories, backdoor
Bashlite	Botnet		Controlled by C&C, carrying out TCP, UDP and HOLD flood attacks
RansomwarePoC	Ransomware	30	Encrypt directories using a generated AES key

Chapter 5

Evaluation

This chapter assesses the dataset created in chapter 4 and compares the differences between the normal behaviour and each malware behaviour of the ElectroSense sensor. This chapter also compares the behaviors between the malware. The malware infected datasets show considerable differences in behavior when compared statistically to one another and also when compared to the normal behavior.

5.1 Results

To start out the evaluation, for each dataset, the mean and standard deviation of the frequency of each system call was calculated. For each 10 second file containing cleaned system call data related to a specific behavior, the mean and standard deviation were calculated. This value was then aggregated to a data frame, where after iterating through all the files of a single malware, the mean and standard deviation were computed on the aggregated data. This results in a mean of means and a standard deviation of standard deviations. Both the mean and standard deviation of the system call frequency was computed with the help of the script *frequency.py*. The script is structured as follows.

Listing 5.1: Frequency.py Setup Snippet

```
31 if(options.input == None):
32     print(parser.usage)
33     exit(0)
34 else:
35     dirs = options.input
36
37 index = []
38 count = 1
39 round = 1
40 df1 = pd.DataFrame()
41 df2 = pd.DataFrame()
42 df_hourWise = pd.DataFrame()
```

- Setup
  This section of the script ensures that all variables needed later on are initialized and that the user has provided a valid input directory, where the cleaned system call data should be located. Listing 5.1 shows an excerpt from the setup section. Lines 31 through 35, ensure that the user specified directory is set. If it is not provided, the script terminates. Lines 38 through 42 initialize multiple variables needed later on. The variable `index` keeps track of the position of the next system call inside of an array, in order to add the values in the corresponding column. The `count` and `round` variables keep record of how many files have been processed in the current folder and how many folder directories have been processed respectively.

- Main Loop
  The main loop, as demonstrated below in Listing 5.2, iterates through each directory specified by the user and subsequently iterates through each sub-directory and its files. It initializes a nested list on line 54, where the first list at `index = 0` represents the system calls found in the files and the list at `index = 1` and upwards, the corresponding frequency value for a specific system call in a concrete file. For example, the values of the first file processed correspond to the list inside the nested list at index = 1 and the n-th file values to the n-th list inside of the nested list. The nested list is reset for every sub-directory, as shown in line 54. Then, a data frame is built for each file and rows that contain `rsync, perf` or `monitoringScri` as process names are removed from the data frame as shown in lines 65 through 67. This is done to ensure that behaviors produced by the *monitoringScript.sh* are not present when calculating the frequency of the system calls of the ElectroSense sensor. Moreover, the "*" sign on multiple system calls, which denotes that that specific system call is a continuation of a previous invocation, is also removed in line 69. Thereafter, the frequencies are calculated for each system call, normalized on line 70 and later on added to the nested list initialized earlier. Lines 76 through 93 handle the addition of the different values to the nested list, in the correct row and column. After all files of a sub-directory are processed, a column, representing the mean of all frequency values for the system calls is added to another data frame that keeps track of all processed directories. Also, the count is increased by one, signifying that a sub-directory has been processed. This procedure is seen through lines 95 and 99.

Listing 5.2: Frequency.py Main Loop

```
46 for dir in dirs:
47     behavior = dir.split("/")[-2]
48     subdirs = [x[0] for x in os.walk(dir)]
49     print(subdirs)
50     for subdir in subdirs[1:]:
51         print("Subdir:",subdir)
52         hour = subdir.split("/")[-1]
53         # create nested list to save values temporarly
54         d = [[],[]]
55         print(subdirs)
56         for files in os.walk(subdir):
57             for filename in files[2]:
```

```
58                  file = os.path.join(subdir,filename)
59                  file = file.replace("\\\\", "\\")
60                  print("Current file {}\n Time: {}".format
                        (file, datetime.datetime.now()))
61                  df = pd.read_csv(file, sep="\t", names=["
                        ABS TIME", "Process Name", "PID", "
                        System_Call"])
62                  df.columns = df.columns.str.replace('\s+'
                        , '_')
63
64                  # remove rsync, perf and monitoringScri
65                  df = df[df['Process_Name'].str.contains("
                        rsync")==False]
66                  df = df[df['Process_Name'].str.contains("
                        perf")==False]
67                  df = df[df['Process_Name'].str.contains("
                        monitoringScri")==False]
68
69                  df['System_Call'] = df['System_Call'].str
                        .replace('*', '')
70                  val_count = df.System_Call.str.split(
                        expand=True).stack().value_counts(
                        normalize=True).sort_index()
71
72                  if(count > len(d)-1):
73                      d.append(['NaN']*len(d[0]))
74                  print("Count: {}, len(d): {}".format(
                        count,len(d)))
75
76                  for key in val_count.keys():
77                      if key not in d[0] and count==1:
78                          d[0].append(key)
79                          index = len(d[0]) - 2
80                          d[1].append(val_count[key])
81
82                      elif key not in d[0] and count!=1:
83                          d[0].append(key)
84                          for i in range(1,len(d)-1):
85                              d[i].append('NaN')
86                          d[count].append(val_count[key])
87
88                      elif key in d[0]:
89                          i = d[0].index(key)
90                          if i > len(d[count])-1:
91                              d[count].append(val_count[key
                                    ])
92                          elif d[count][i] == 'NaN':
```

```
93                                  d[count][i] = val_count[key]
94                      count+=1
95                  df = pd.DataFrame(d[1:], columns=d[0])
96                  mean = df.mean()
97                  mean.name = '{}'.format(hour)
98                  df_hourWise = pd.concat([df_hourWise, mean],
                        axis=1)
99                  count = 1
```

- Mean
  After the main loop is done processing all directories and sub-directories, the mean of each system call on a per directory basis is concatenated onto the final data frame as shown in Listing 5.3.

- Standard Deviation
  Right after the mean has been computed and stored onto a mean data frame, the standard deviation, as depicted in Listing 5.4 is also computed and stored onto a standard deviation data frame.

- Output
  Both the mean and standard deviation data frames are then saved to corresponding comma separated value (CSV) files.

Listing 5.3: Frequency.py & Ngram.py Mean Section

```
105     mean = df_hourWise.mean()
106     mean.name = '{}'.format(behavior)
107     if round == 1:
108         df1 = pd.DataFrame(mean)
109     else:
110         df1 = pd.concat([df1, mean], axis=1)
```

Listing 5.4: Frequency.py & Ngram.py Standard Deviation Section

```
115     std = df_hourWise.std(axis=1)
116     std.name = '{}'.format(behavior)
117     df2 = pd.concat([df2, std], axis=1)
118     round+=1
```

At the same time, the sequence of the system calls was also examined. For the purpose of this work, all possible 2-grams, also referred to as bigrams, were computed and thereafter followed the same computing process as with the frequency of system calls. The script *ngram.py* was created to accomplish this task. *Ngram.py* follows a similar pattern to *frequency.py* and is arranged as follows.

Listing 5.5: Ngram.py Setup Section

```
29 parser.add_option('-n', '--ngram', dest = 'n',
30                     type = 'int',
31                     help = 'specify n-gram length')
```

```
32 (options, args) = parser.parse_args()
33 if(options.input == None):
34     print(parser.usage)
35     exit(0)
36 else:
37     dirs = options.input
38
39 if(options.n == None):
40     print(parser.usage)
41     exit(0)
42 else:
43     n = options.n
```

- Setup
  A snipped of the source code is displayed in Listing 5.5. Although similar to the *frequency.py* setup section, the *ngram.py* script takes an extra parameter "n" which the user needs to specify. This parameter dictates the size of the n-grams calculated.

- Main Loop
  Is almost identical to the main loop of the *frequency.py* script. However lines 82 through 85, shown in Listing 5.6, creates the bigrams and calculates the frequency of each.

Listing 5.6: Ngram.py Main Loop Section Snippet

```
82                  words = (df.System_Call.str.split().
                        explode())
83                  n_gram = ngrams(words, n)
84                  n_gram_df = pd.DataFrame(n_gram)
85                  val_count = n_gram_df.value_counts(
                        normalize=True)
```

- Mean
  Works analog to the mean section of *frequency.py* and Listing 5.3.

- Standard Deviation
  This section, tasked with calculating the standard deviation of the mean frequencies of all bigrams is the same as the one presented in Listing 5.4.

- Output
  Outputs both the means and standard deviation data frames for the desired n-grams onto CSV files.

After having used both scripts to compute multiple statistics, the next subsections below, put each malware behavior into comparison with the normal behavior by first comparing the means and standard deviation of system call frequencies. Thereafter, to provide a complete comparison and a broader perspective, the mean frequency of the bigrams for each behavior are compared to the normal behavior. Finally, a contrast between all anomalous

behaviors is drawn by comparing each malware's behavior system call bigram frequency. Furthermore, the system call `nanosleep` and the bigram `nanosleep-nanosleep` have been omitted by all plots presented below. This, in order to have a clearer view of the finer margins presented between all other system calls and bigrams below, as `nanosleep` and `nanosleep-nanosleep` posses high frequency percentages, since it is the main system call used by the ElectroSense sensors standard behavior.

5.1.1 Thetick

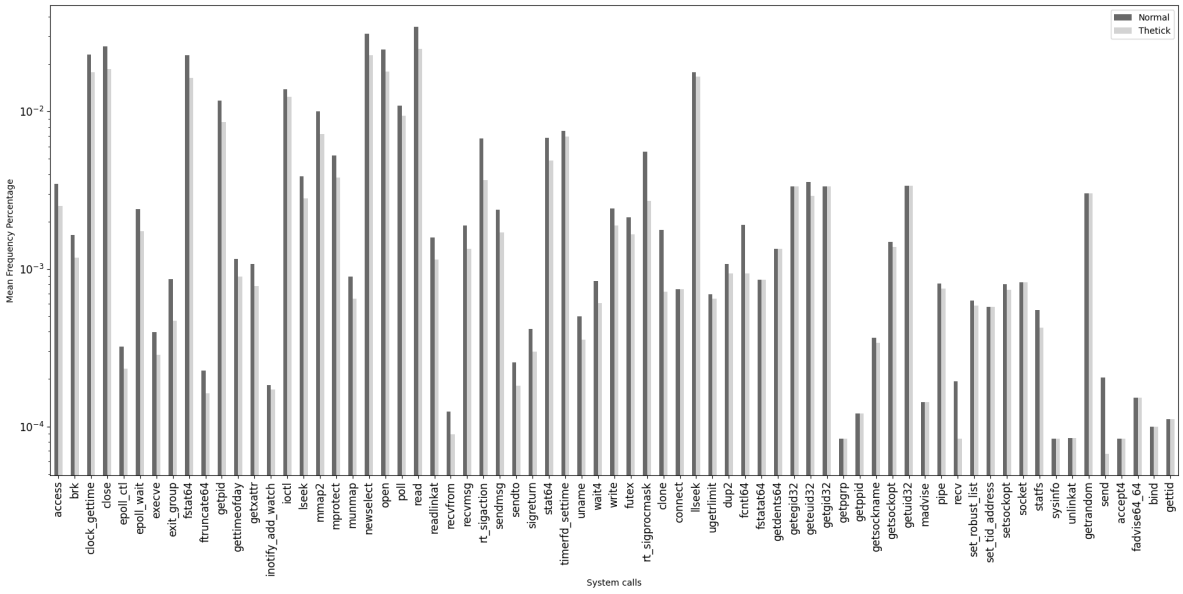


Figure 5.1: Mean Values between Normal and Thetick behavior

Thetick malware makes use of multiple system calls. Among them are `recv, send, write, close, getsockopt, clone` and more. Figure 5.1 depicts the frequency means of each system call found in normal and during Thetick attack behavior. The x-axis corresponds to the system calls found in both behaviors and the y-axis shows the mean frequency percentage. Most system calls have a similar frequency mean so not much can be read into Figure 5.1, other than that at a first look both behavior operate similarly on the device. However, considering the standard deviation of the system call frequencies, portrayed in Figure taking a closer look to Figure 5.2, which instead of the mean of the frequencies, portrays the standard deviation of them, a clearer difference between behaviors can be distinguished. Looking at crucial system calls used by Thetick malware, in particular `send, set_robust_list, fnctl64` and `getsockopt`, they experience an increase of 1434.45%, 1488.84%, 217.862% and 1360.88% respectively, which indicates a higher scattering of the amount of times and the consistency with which these system calls were summoned while the device was infected with Thetick malware.

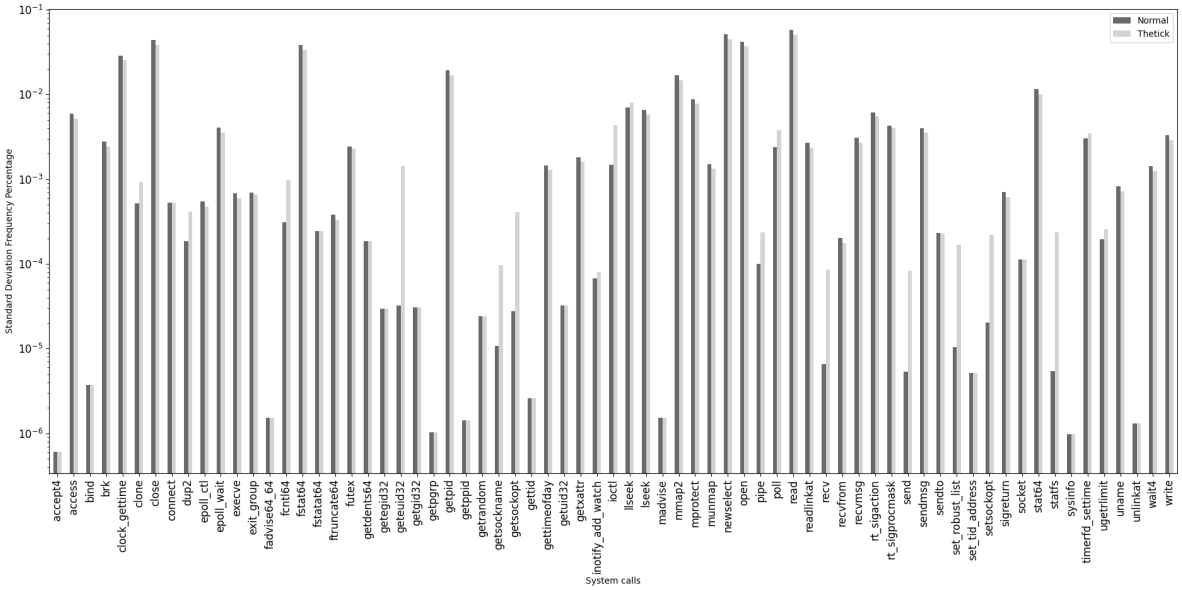


Figure 5.2: Standard Deviation Frequency between Normal and Thetick behavior

In addition, Figure 5.3 outlines the mean frequency of various bigrams found in either the normal behavior dataset, Thetick behavior dataset or both. It displays multiple bigrams which are only present while the device is infected with thetick backdoor malware. A few examples are `(nanosleep-recv)`, `(send-send)` and `(nanosleep-write)`. Also, some bigrams have marginally higher means in Thetick behavior. For instance, `(nanosleep-execve)`, `(nanosleep-recvmsg)`, `(nanosleep-getpid)` and `(write-nanosleep)` all have higher frequency means in Thetick behavior, implying a higher usage of these system calls.

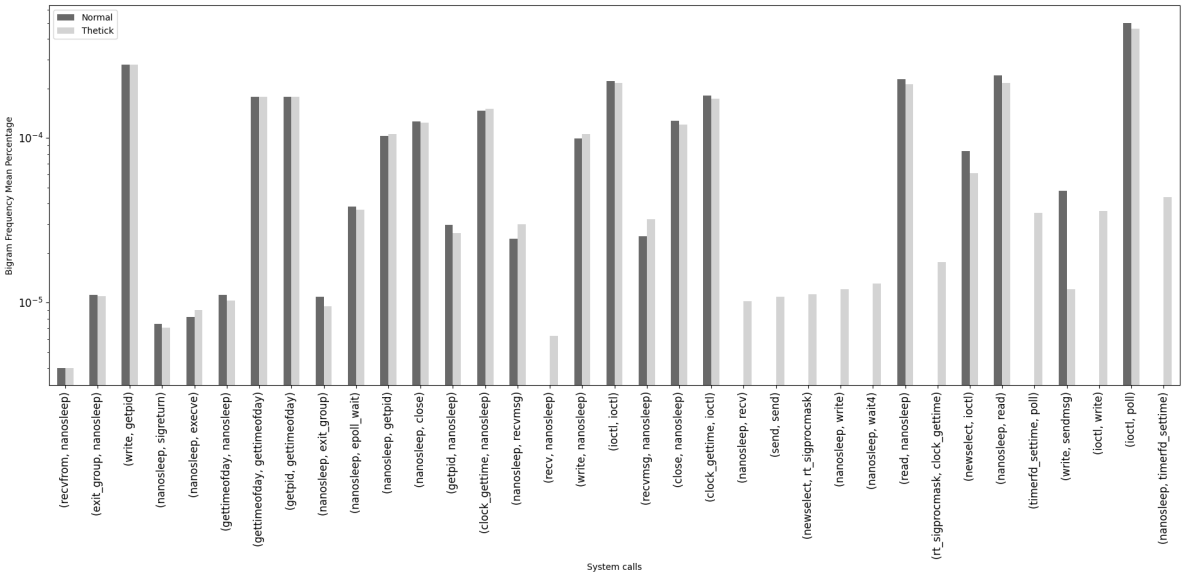


Figure 5.3: System Call Bigram Frequency: Normal and Thetick behavior

5.1.2 Bashlite

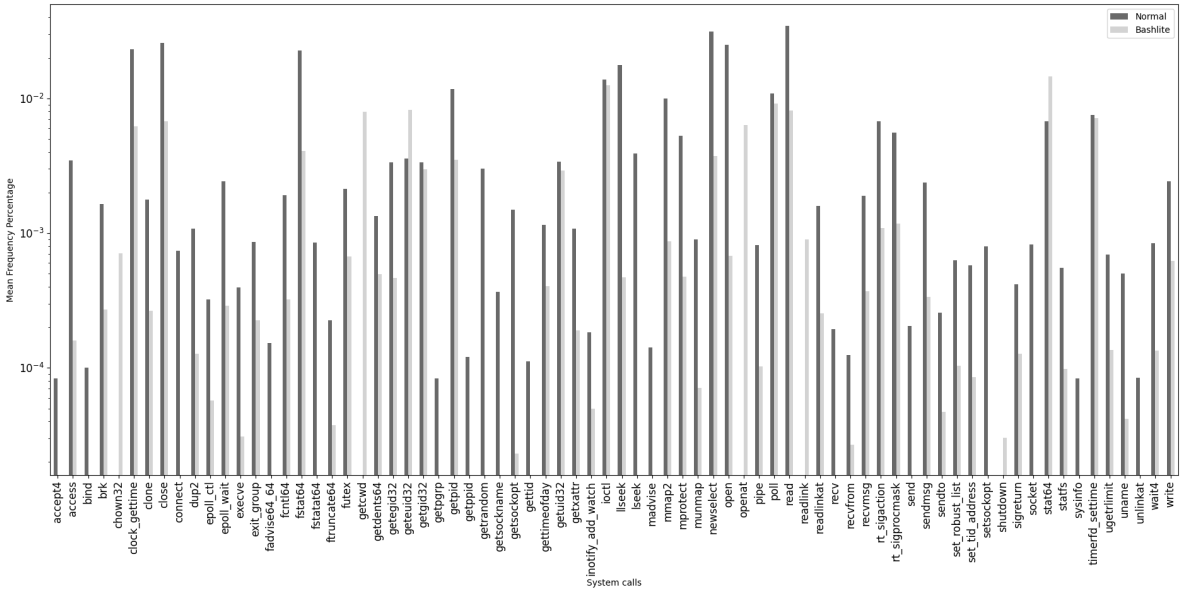


Figure 5.4: Mean Values between Normal and Bashlite behavior

Figure 5.4 displays the mean frequency values for each system call for both normal and Bashlite behavior. A clear trend where the normal behavior has higher means for almost each system call with some exceptions can be observed. Yet, evaluating a couple of key system calls used by the Bashlite malware, the difference in behavior between the two scenarios is noticeable. For example `chown32, getcwd, openat` and `readlinkat` are all not present during normal behavior. Thus, suggesting that these system calls are only used when the device is under the attack of the bashlite botnet. Additionally, system calls `stat64` and `geteuid32`, both used by the Bashlite botnet, have higher usage frequencies, 113.878% and 128.488% increase respectively, compared to the normal operational behavior of the device.

Moreover, taking a look at the standard deviation of the frequency of each system call in Figure 5.5, other system calls used by both normal and bashlite behavior occupy a higher standard deviation when the device is under attack compared to when it is not under attack. Also, system calls that have the value zero for its frequency of one of both, normal or bashlite behavior, in Figure 5.4 and have a standard deviation of zero in Figure 5.5, like for example `recv`, had a constant frequency value across the device's monitoring phase while under attack of the Bashlite botnet.

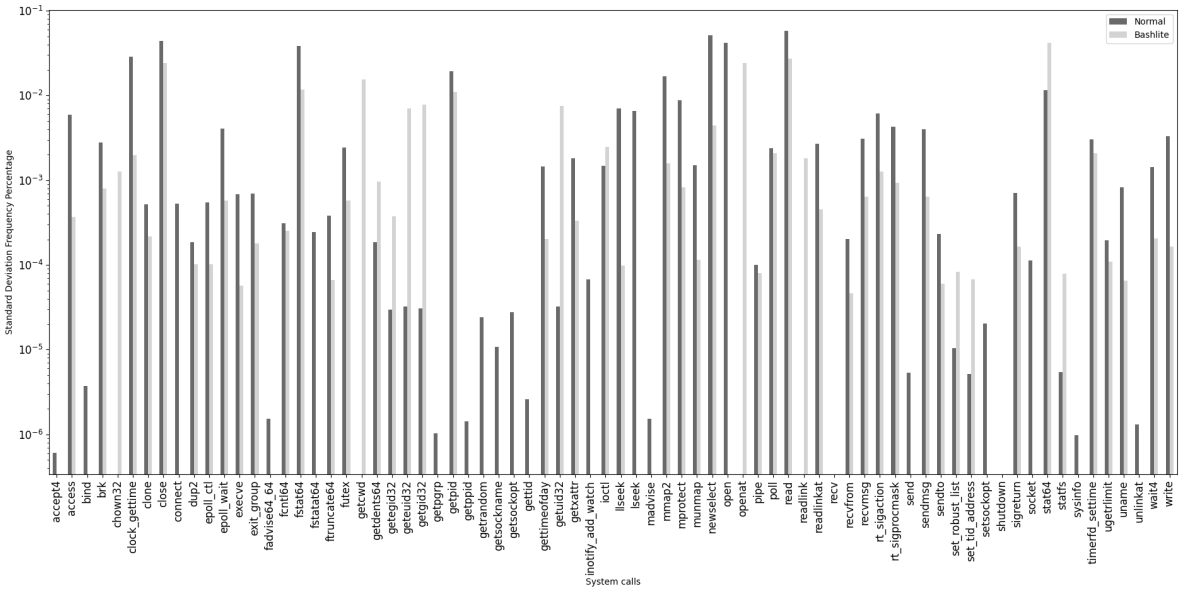


Figure 5.5: Standard Deviation Frequency between Normal and Bashlite behavior

Next, analyzing the 2-gram frequencies for both behaviors, the difference in behaviors is more apprehensible. Figure 5.6 demonstrates the top 35 bigrams, ordered by mean frequency percentage differential between both behaviors in ascending order. Numerous bigrams, `(gettimeofday-nanosleep)`, `(write-nanosleep)`, `(close-nanosleep)` and `(nanosleep-close` among others, have a higher frequency for when the device is under control of the Bashlite botnet. Similarly, bigrams such as `(futex-openat)`, `(nanosleep-wait4)`, `(brk-openat)`, `(uname-mmap)`, `(stat64-nanosleep)` and `(sendmsg-openat)` to name a few, only exist for when the device is being used by the Bashlite malware. The bigram figure further proves the asymmetry of system call usage between normal and bashlite behavior.

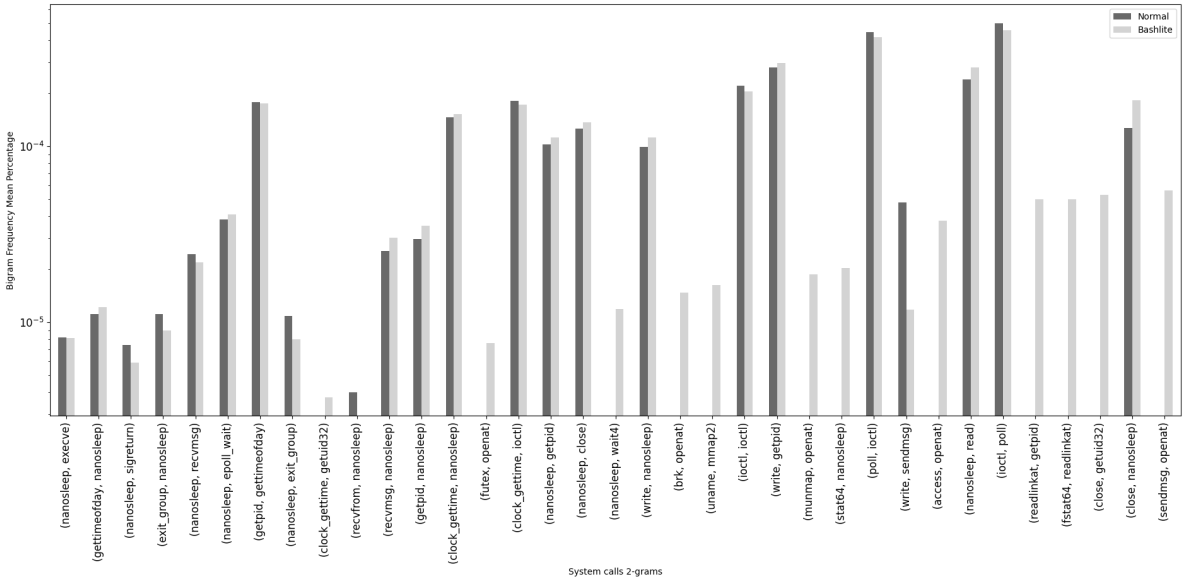


Figure 5.6: System Call Bigram Frequency: Normal and Bashlite behavior

5.1.3 Bdvl

Moving on to the Bdvl rootkit, the means of each system call frequencies presented in Figure 5.7 illustrates the disparity between both normal and Bdvl behavior. A careful examination on a couple of key system calls that underline the influence of the Bdvl rootkit malware on the device, for instance `stat64` and `send`, which had a 266% and a 948.837% increase in average frequency respectively, highlights the differences in system call invocation between both behaviors. Additionally, system calls such as `chown32, getcwd, readlink` and `openat` are only present when the device is being attacked by the rootkit.

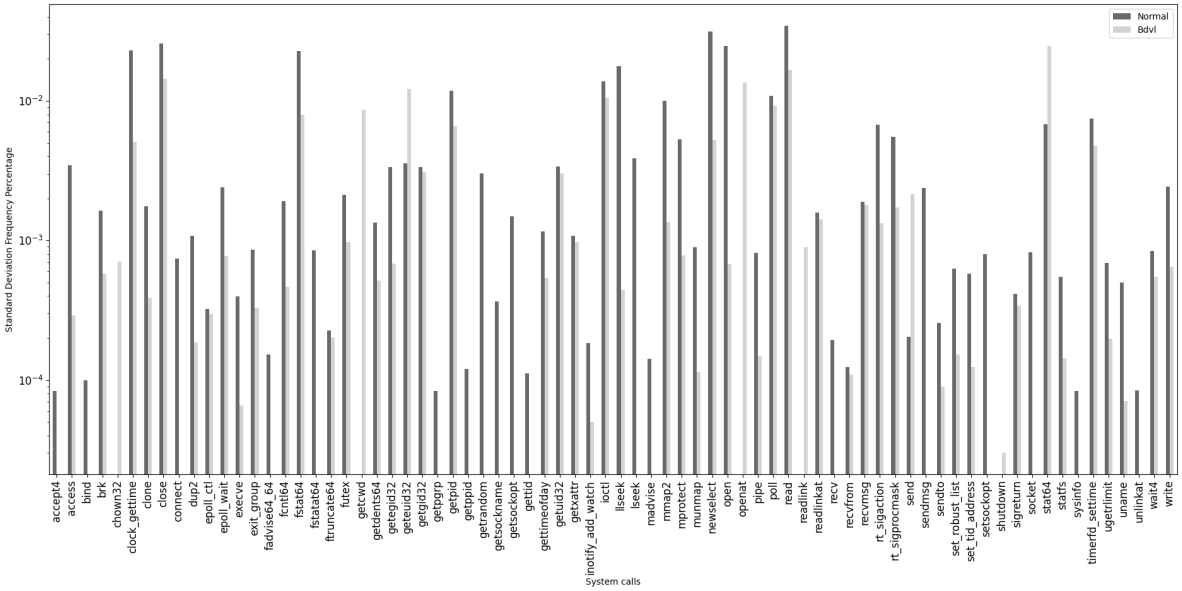


Figure 5.7: Mean Values between Normal and Bdvl behavior

In parallel, some of the crucial system calls used by the rootkit such as, `stat64, getuid32` and `llseek`, also display a higher standard deviation while the device is under a Bdvl attack. This means that the execution of these system calls is scattered more widely across the Bdvl dataset in comparison with the normal behavior dataset. Which is explained through the fact that the rootkit is not invoking system calls synchronously, but instead reacting to when for example new data is created on the system in order to steal it.

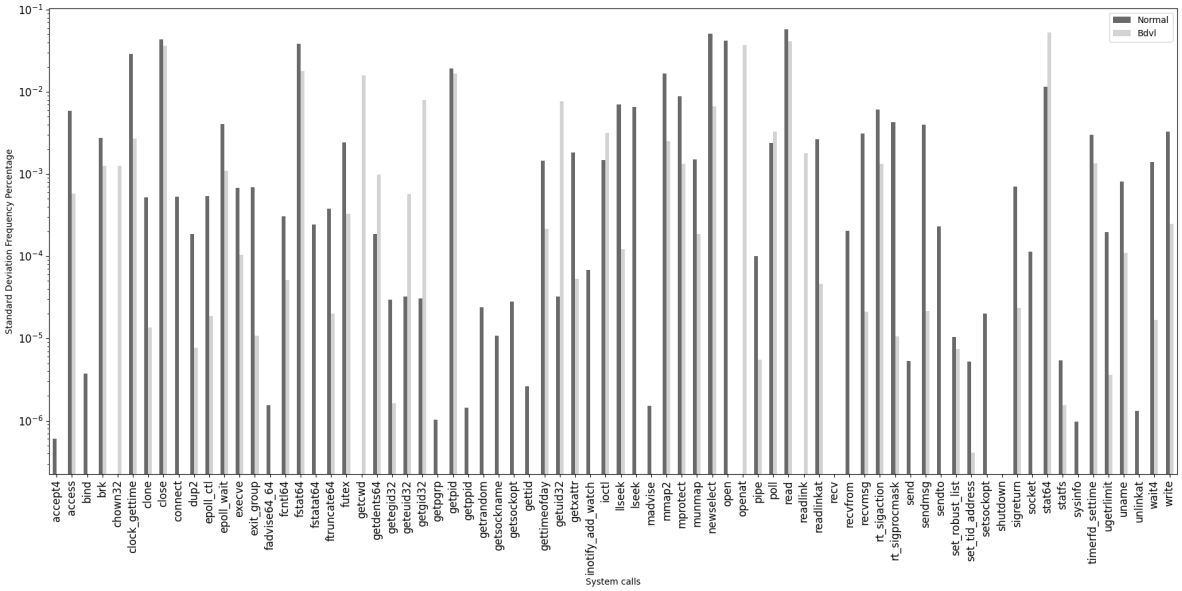


Figure 5.8: Standard Deviation Frequency between Normal and Bdvl behavior

The system calls gathered while the device was under the attack of the Bdvl malware

produced over 368 bigrams. When put in comparison with the bigrams from the normal behavior, the distinction between normal behavior and behavior under Bdvl rootkit attack is unambiguous. Figure 5.9 shows the top 35 2-gram system calls sequences ordered by biggest frequency difference between normal and Bdvl behavior. Multiple bigrams such as `(chown32-openat)`, `(chown32-access)`, `(openat-fstat64)` and `(access-gettimeofday)`, are not present in the system call bigrams of the device under normal behavior. These bigrams disparities further highlight the difference between normal and Bdvl internal behavior of the device.

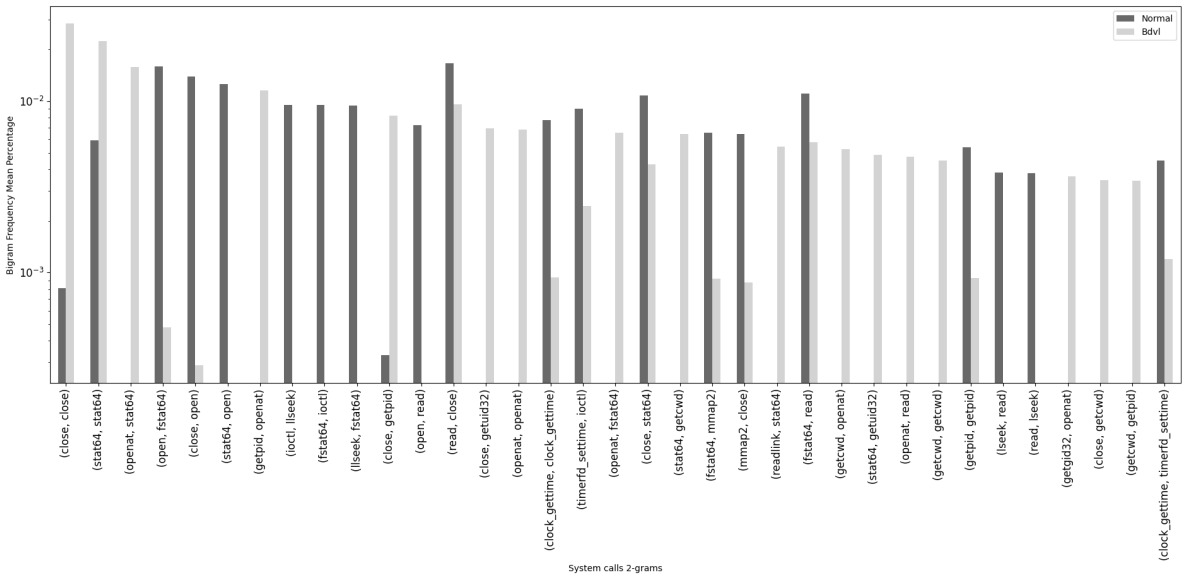


Figure 5.9: System Call Bigram Frequency: Normal and Bdvl behavior

5.1.4 RansomwarePoC

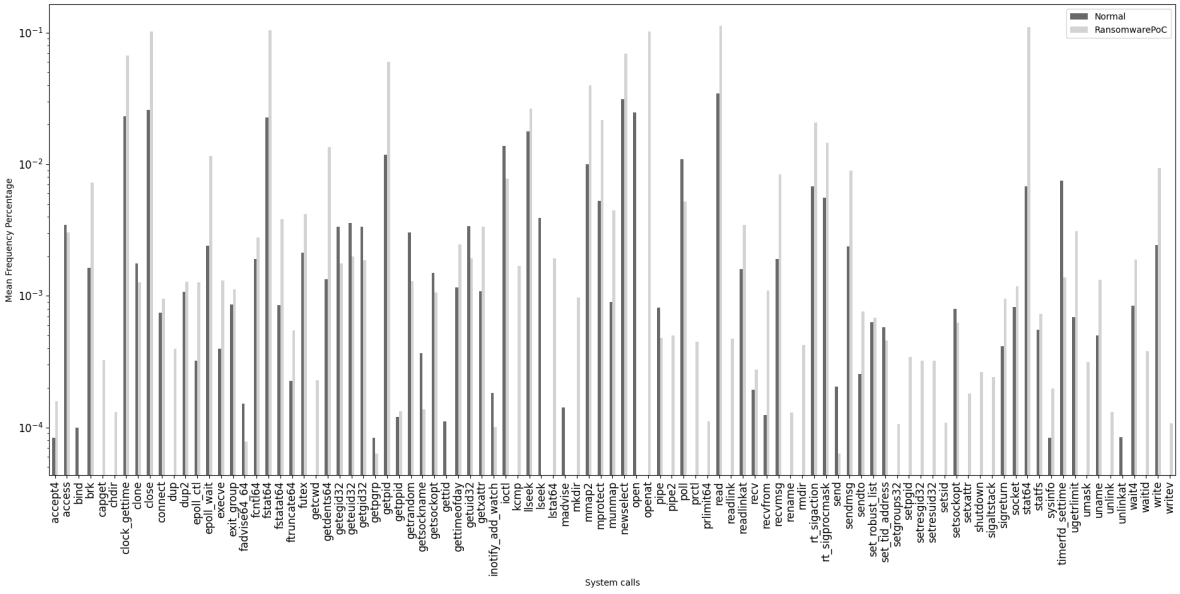


Figure 5.10: Mean Values between Normal and RansomwarePoC behavior

Figure 5.10 delivers a distinct distribution of the mean of system calls for both RansomwarePoC malware and normal behavior. The RansomwarePoC malware introduces multiple system calls that would not normally be solicited by the ElectroSense sensor's normal behavior. A couple of these system calls are `capget, dup, kcmp, lstat64, prctl, prlimit64, unmask, unlink` and others. Also, the RansomwarePoC exploits system calls used in the normal behavior substancially more, as almost all system call means are higher for the RansomwarePoC behavior. Some of these system calls are `getdents64, rt_sigacation` and `munmap` with an increase of 903.937%, 205.567% and 400.243% respectively over the normal behavior mean frequency.

Further, examining the standard deviation comparison between the two behaviors, Figure 5.11 shows a trend in which RansomwarePoC based system call frequencies have lower standard deviations than the normal behavior counter parts. This could partially be due to the fact that multiple files where encrypted constantly throughout the monitoring phase. Thus, implying that the system call mean frequencies were stabler when the device was infected with the RansomwarePoC malware.

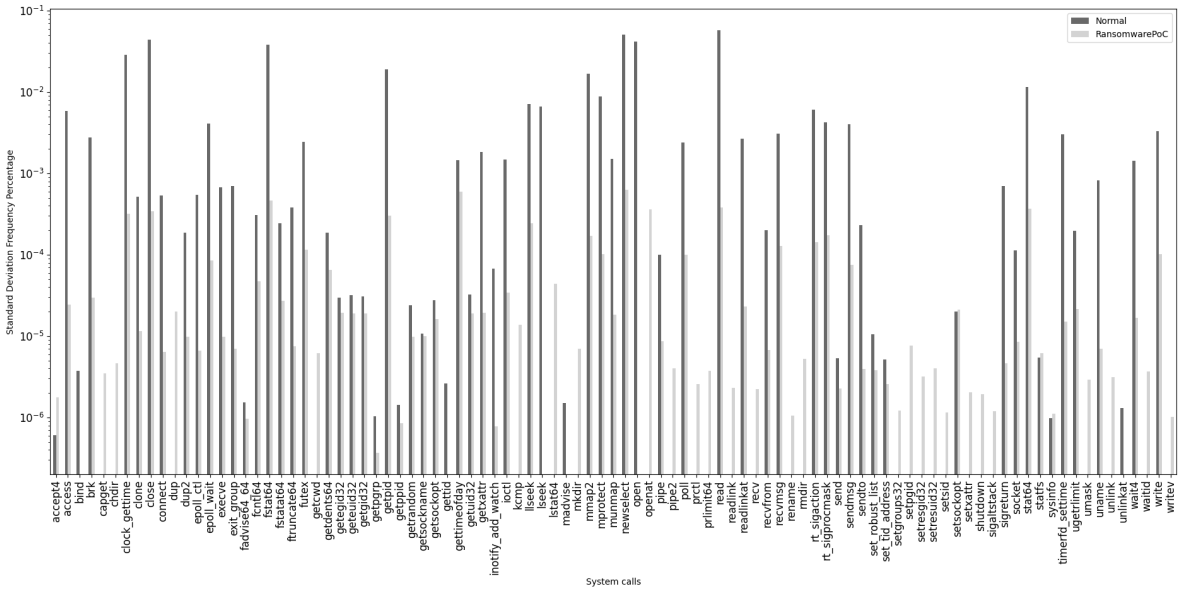


Figure 5.11: Standard Deviation Frequency between Normal and RansomwarePoC behavior

Moreover, both datasets, normal and ransomware behavior, produced 541 system call bigrams combined. Figure 5.12 displays the top 35 bigrams, ordered by absolute difference of a bigram between both datasets. Figure 5.12 emphasizes through bigrams such as `(openat-fstat64)`, `(read, close)`, `(fstat64-llseek)`, `(fstat64-mmap2)` and `(mmap-mprotect)`, the difference of the device's internal behavior when operating normally compared to when being attacked by RansomwarePoC malware.

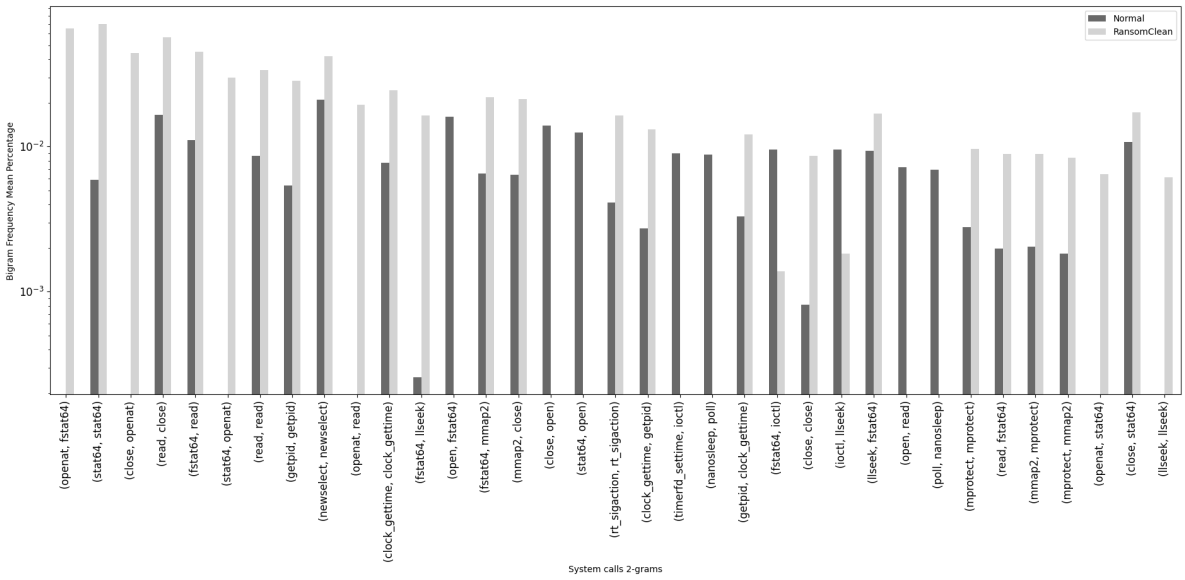


Figure 5.12: System Call Bigram Frequency: Normal vs. RansomwarePoC behavior

5.1.5 Comparing Malware

Finally, there is also a meaningful difference between the malware behaviors executed and discussed in this thesis. Figure 5.13 exemplifies 35 bigram mean frequencies across all the different types of malware used in this thesis. Thetick for example, is the only malware that contains the system call bigram trace `(lseek-read)`. Bashlite for instance, has the highest frequency in the bigrams `(iotl-timerfd_settime)`, `(poll-nanosleep` and `(close-nanosleep)`. Moving on to the Bdvl malware device's induced behavior, it contains the highest frequency of the bigram `(poll- ioctl)`, `(nanosleep-poll)` and `(nanosleep-ioctl)`. Finally, RansomwarePoC also has multiple bigrams for which the frequency is the highest compared to other malware. Some examples are `(read-close)`, `(read-read)`, `(gettimeofday-write)` and `(fstat64-read)`.

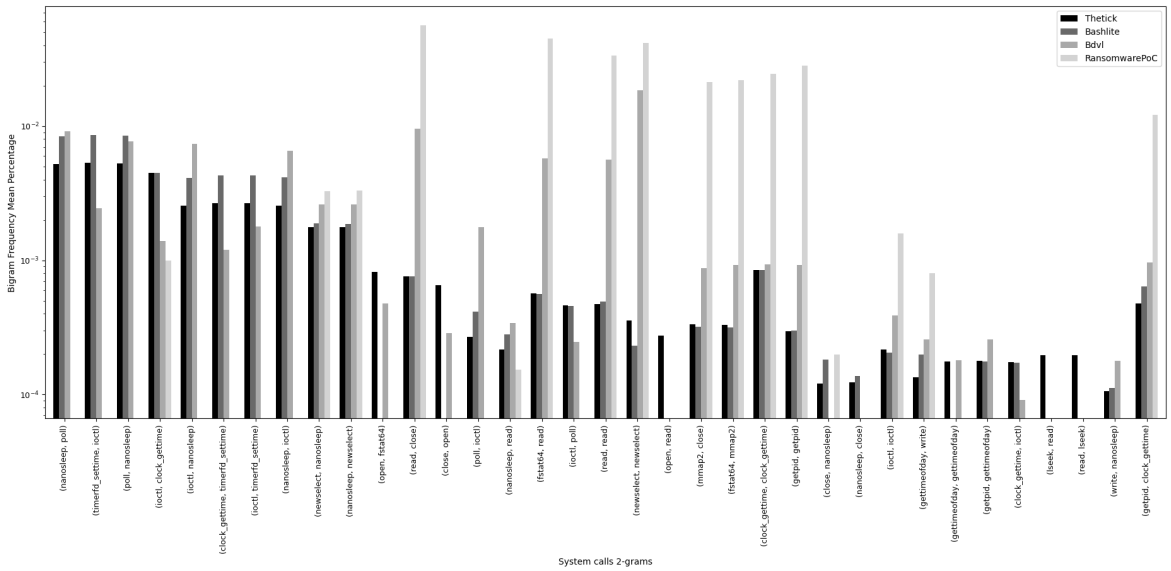


Figure 5.13: System Call Bigram Frequency: Thetick vs. Bashlite vs. Bdvl vs. RansomwarePoC

Chapter 6

Summary, Conclusions and Future Work

6.1 Summary and Conclusions

This chapter summarizes the intended goal of this thesis and provides a conclusion to the achieved dataset.

The main goal of this thesis was to create a dataset that models the internal behavior of an ElectroSense sensor while it is under attack of specific malware by gathering system calls. This dataset was broken down into multiple datasets, each representing a different type of behavior. Five datasets were created. Each representing one of the following four behaviors: normal, under control of a Bashlite botnet, Thetick backdoor, Bdvl rootkit and RansomwarePoC.

In order to generate these datasets, a monitoring script was created that gathered all system calls of the target device. The target device in this thesis was an ElectroSense sensor, based on a Raspberry Pi 4. After the device was set up and registered as part of the ElectroSense network, the next step was to gather system calls of the normal behavior of the embedded device. Thereafter, the task was to set up real life environments for each malware and execute them on the Raspberry Pi and gather the system calls simultaneously. Here, for the Bashlite malware, a remote C&C server and a designated target device were initialized. After infecting the ElectroSense sensor with the Bashlite malware and adding it to the botnet, the Raspberry Pi started executing flooding attacks on the designated remote target. After the system call gathering process was done, the next phase was to infect the device with Thetick backdoor and execute commands on the Raspberry Pi through the backdoor. Because Thetick backdoor provides a C&C server, this was also initialized to take full advantage of the capabilities of the malware. Next, the device was infected with the Bdvl rootkit. Through the rootkit spawned backdoor multiple commands were also executed on the device. Finally, the RansomwarePoc was installed on the device and multiple directories were encrypted. The system call gathering process took place for each malware and also the normal behavior of the sensor. Also important to note, before transitioning to a new malware infection phase, the past malware was completely removed from the device first. Once the data was gathered for each scenario, the data was then cleaned to produce the finalized datasets. Each dataset contains 10

second interval files with system calls. These files are stored in a per-date and then per-hour directories. The features found in all datasets are the absolute time at which a specific system call was executed, the name of the process that initialized the system call, the PID and the system call name itself.

The final step in this thesis was to evaluate the finalized datasets statistically. For this the frequency and sequences of system calls were considered statistically. For the frequency of system calls, the mean and standard deviation of each system call in each dataset was computed. Theses statistical values were compared with the statistical values of the normal behavior and the results plotted to be able to visualize the differences. In conclusion, the changes in behavior were noticeable when comparing frequency mean and standard deviation. Additionally, the statistical analysis of the sequences of system calls, which was done by computing the mean of bigrams, delivered a richer and clearer comparison between different infected behavior and normal behavior. Each of the four behaviors under attack showcased a significant statistical difference when compared to the normal behavior. Also when comparing the malware behaviors with each other, a notable difference in system call invocation was identified when comparing their bigrams. Thus showing the un-explored potential that can be taken advantage of by feeding the dataset to ML or DL algorithms to better the accuracy of HIDSs.

6.2 Future Work

The focus of this thesis was laid on monitoring and gathering system calls one malware attack vector per malware family. There are multiple other malware attack vectors affecting IoT devices which can be used to construct an even more precise depiction of the internal behavior of the ElectroSense sensor if also these other attack vectors are taken into account. An example of another botnet that could be considered is the Mirai botnet, which also has affected various IoT device in recent years. This thesis also utilized a LD_PRELOAD rootkit in Bdvl. Other LD_PRELOAD rootkits such as Beurk and also LKM rootkits including, Diamorphine could also be taken into account. The source code of all these aforementioned attack vectors can be found online, are listed in Table 2.2 and could be used in future work.

At the same time, future work could call for the monitoring of multiple ElectroSense sensors at the same time, in order to create a bigger dataset, as this thesis focused on capturing the internal behavior of a single ElectroSense sensor. Also, an interesting future work scenario would be to employ the sensor as the target node of a DDoS attack initialized through a botnet, in order to better understand the behavior of the device when it is being flooded through TCP, UDP or other protocols, since in this thesis the ElectroSense sensor was used as the attack bot in the botnet.

Bibliography

[1] Lu, Y., & Da Xu, L. (2018). Internet of things (iot) cybersecurity research: A review of current research topics. IEEE Internet of Things Journal, 6(2), 2103-2115

[2] Pan, J. & Yang, Z. Cybersecurity challenges and opportunities in the new" edge computing+ IoT" world. *Proceedings Of The 2018 ACM International Workshop On Security In Software Defined Networks Network Function Virtualization*. pp. 29-32 (2018)

[3] *What is a Raspberry Pi?*, `https://opensource.com/resources/raspberry-pi`, (Accessed on 03.04.2022)

[4] Malik, J. *Cloud amp; IoT; Or, How I Learned To Stop Worrying About Security amp; Love Innovation*. (2017,3), https://cdn-cybersecurity.att.com/docs/analyst-reports/rsa-2017-report.pdf

[5] Khraisat, A., Gondal, I., Vamplew, P. & Kamruzzaman, J. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity*. **2**, 20 (2019,7), `https://doi.org/10.1186/s42400-019-0038-7`

[6] Bezawada, B., Bachani, M., Peterson, J., Shirazi, H., Ray, I. & Ray, I. Behavioral Fingerprinting of IoT Devices. *Proceedings Of The 2018 Workshop On Attacks And Solutions In Hardware Security*. pp. 41-50 (2018), `https://doi.org/10.1145/3266444.3266452`

[7] *System calls: What are system calls and why are they necessary?*, `https://www.ionos.com/digitalguide/server/know-how/what-are-system-calls/`, (Accessed on 03.02.2022)

[8] Ngo, M.V., Chaouchi, H., Lou, T., & Quek, T. Q. (2020). Adaptive anomaly detection for IoT data in hierarchical edge computing. arXiv prepint arXiv:2001.03314 Sources, Techniques, Application Scenarios, and Datasets.

[9] Jeon, J., Park, J.H., & Jeong, Y.S. (2020). Dynamic Analysis for IoT Malware Detection with Convolution Neural Network model. IEEE Access

[10] Rhode, M., Burnap, P., & Jones, K. (2018). Early-stage malware prediction using recurrent neural networks. computers security, 77, 578-594.

[11] Kerrisk, M., *syscalls(2) â Linux manual page*, `https://man7.org/linux/man-pages/man2/syscalls.2.html` (Accessed on 10.12.2021)

[12] Shoemaker, A., *How to Identify a Mirai-Style DDoS Attack*, https://www.imperva.com/blog/how-to-identify-a-mirai-style-ddos-attack (Accessed on 13.01.2022)

[13] *Backdoor computing attacks*, https://www.malwarebytes.com/backdoor (Accessed on 04.04.2022)

[14] Hashemi, S. & Zarei, M. Internet of Things backdoors: resource management issues, security challenges, and detection methods. *Transactions On Emerging Telecommunications Technologies*. **32**, e4142 (2021)

[15] Humayun, M., Jhanjhi, N., Alsayat, A. & Ponnusamy, V. Internet of things and ransomware: Evolution, mitigation and prevention. *Egyptian Informatics Journal*. **22**, 105-117 (2021), https://www.sciencedirect.com/science/article/pii/S1110866520301304

[16] Yaqoob, I., Ahmed, E., Rehman, M., Ahmed, A., Al-garadi, M., Imran, M. & Guizani, M. The rise of ransomware and emerging security challenges in the Internet of Things. *Computer Networks*. **129** pp. 444-458 (2017), https://www.sciencedirect.com/science/article/pii/S1389128617303468, Special Issue on 5G Wireless Networks for IoT and Body Sensors

[17] Nemeth, K., Buttyan, L. & Papp, D. Detection of persistent rootkit components on embedded IoT devices. (2020)

[18] Thakur, A. Memory Malware Part 0x2 â Crafting LD_PRELOAD Rootkits in Userland. *Medium.com*. (2020,5), https://compilepeace.medium.com/memory-malware-part-0x2-writing-userland-rootkits-via-ld-preload-30121c8343d5

[19] Sanchez, P.M.S., Valero, J.M.J., Celdran, A. H., Bovet, G., Perez, M.G., & Perez, G.M. (2020). A Survey on Device Behavior Fingerprinting: Data Sources, Techniques, Application Scenarios, and Datasets.

[20] Zhou, L. & Makris, Y. Hardware-based on-line intrusion detection via system call routine fingerprinting. *Design, Automation Test In Europe Conference Exhibition (DATE), 2017*. pp. 1546-1551 (2017)

[21] Liao, Y. & Vemuri, R. Using Text Categorization Techniques for Intrusion Detection. (2002,7)

[22] University of New Mexico Computer Science Department, Farris Engineering Center. Computer immune systems - data sets and software. 1999, (https://www.cs.unm.edu/~immsec/systemcalls.html) (Accessed on 19.03.2022)

[23] Kolias, C., Kambourakis, G., Stavrou, A. & Voas, J. DDoS in the IoT: Mirai and other botnets. *Computer*. **50** pp. 80-84 (2017,1)

[24] Marzano, A., Alexander, D., Fonseca, O., Fazzion, E., Hoepers, C., Steding-Jessen, K., Chaves, M., Cunha, Ã., Guedes, D. & Meira, W. The Evolution of Bashlite and Mirai IoT Botnets. *2018 IEEE Symposium On Computers And Communications (ISCC)*. pp. 00813-00818 (2018)

[25] hammerzeit. Bashlite. 2016, `https://github.com/hammerzeit/BASHLITE` (Accessed on 15.12.2021)

[26] unix-thrust. Beurk. 2017, `https://github.com/unix-thrust/beurk` (Accessed on 15.12.2021)

[27] Error996. Bdvl. 2020, `https://github.com/Error996/bdvl` (Accessed on 15.12.2021)

[28] m0nad. Diamorphine. 2013, `https://github.com/m0nad/Diamorphine` (Accessed on 08.11.2021)

[29] jgamblin. Mirai. 2016, `https://github.com/jgamblin/Mirai-Source-Code` (Accessed on 15.12.2021)

[30] SkryptKiddie. httpBackdoor. 2020, `https://github.com/SkryptKiddie/httpbackdoor` (Accessed on 20.12.2021)

[31] jakoritarleite. Backdoor. 2018, `https://github.com/jakoritarleite/backdoor` (Accessed on 20.12.2021)

[32] nccgroup. The tick. 2020, `https://github.com/nccgroup/thetick` (Accessed on 15.12.2021)

[33] *EKANS Ransomware and ICS Operations*, `https://www.dragos.com/blog/industry-news/ekans-ransomware-and-ics-operations/` (Accessed on 07.12.2021)

[34] jimmy-ly00. Ransomware-PoC. 2020, `https://github.com/jimmy-ly00/Ransomware-PoC` (Accessed on 15.12.2021)

[35] Gupta, B., Misra, M. & Joshi, R. An ISP level Solution to Combat DDoS attacks using Combined Statistical Based Approach. *International Journal Of Information Assurance And Security (JIAS).* **3** (2012,3)

[36] Gurkok, C. Chapter 41 - Cyber Forensics and Incidence Response. *Computer And Information Security Handbook (Third Edition).* pp. 603-628 (2017), `https://www.sciencedirect.com/science/article/pii/B9780128038437000417`

[37] Le, H., Ngo, Q. & Le, V. Iot Botnet Detection Using System Call Graphs and One-Class CNN Classification. *VOLUME-8 ISSUE-10, AUGUST 2019, REGULAR ISSUE.* (2019)

[38] Dymshits, M., Myara, B. & Tolpin, D. Process monitoring on sequences of system call count vectors. (2017,10), `https://www.ijitee.org/wp-content/uploads/papers/v8i10/J90910881019.pdf`

[39] Phu, T., Dang, K., Quoc, D., Tho, N. & Binh, N. A Novel Framework to Classify Malware in MIPS Architecture-Based IoT Devices. *Security And Communication Networks.* **2019** pp. 1-13 (2019,12)

[40] Warrender, C., Forrest, S. & Pearlmutter, B. Detecting intrusions using system calls: alternative data models. *Proceedings Of The 1999 IEEE Symposium On Security And Privacy (Cat. No.99CB36344)*. pp. 133-145 (1999)

[41] Mpanti, A., Nikolopoulos, S. & Polenakis, I. Malicious Software Detection and Classification utilizing Temporal-Graphs of System-call Group Relations. *CoRR*. **abs/1812.10748** (2018), http://arxiv.org/abs/1812.10748

[42] Grimmer, M., RÃ¶hling, M., Kricke, M., Franczyk, B. & Rahm, E. Intrusion Detection on System Call Graphs. (2018,2)

[43] Hashemi, S. & Zarei, M. Internet of Things backdoors: Resource management issues, security challenges, and detection methods. *Transactions On Emerging Telecommunications Technologies*. pp. 25 (2021,2), https://onlinelibrary.wiley.com/doi/epdf/10.1002/ett.4142

[44] Lippmann, R., Haines, J., Fried, D., Korba, J. & Das, K. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks*. **34** pp. 579-595 (2000,10), https://archive.ll.mit.edu/ideval/files/1999Eval-ComputerNetworks2000.pdf

[45] Lincoln Laboratory, Massachussetts Institute of Technology: 1999 Darpa Intrusion Detection Evaluation Dataset. https://www.ll.mit.edu/r-d/datasets/1999-darpa-intrusion-detection-evaluation-dataset (Accessed 27.12.2021)

[46] Haines, J., Lippmann, R., Fried, D., Zissman, M. & Tran, E. 1999 DARPA Intrusion Detection Evaluation: Design and Procedures. (2001,2), https://apps.dtic.mil/sti/pdfs/ADA387747.pdf

[47] Brown, C., Cowperthwaite, A., Hijazi, A. & Somayaji, A. Analysis of the 1999 DARPA/Lincoln Laboratory IDS evaluation data with NetADHICT. (2009,8).https://www.ccsl.carleton.ca/paper-archive/brown-cisda-09.pdf

[48] Tavallaee, M., Bagheri, E., Lu, W. & Ghorbani, A. A detailed analysis of the KDD CUP 99 data set. *IEEE Symposium. Computational Intelligence For Security And Defense Applications, CISDA*. **2** (2009,7)

[49] Stolfo, S., Fan, W., Lee, W., Prodromidis, A. & Chan, P. Cost-based modeling for fraud and intrusion detection: results from the JAM project. *Proceedings DARPA Information Survivability Conference And Exposition. DISCEX'00*. **2** pp. 130-144 vol.2 (2000)

[50] KDD Cup 1999 dataset. 1999, http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html (Accessed on 25.12.2021)

[51] University of New Brunswick. NSL-KDD dataset. 2009, https://www.unb.ca/cic/datasets/nsl.html (Accessed on 29.12.2021)

[52] Sharafaldin, I., Habibi Lashkari, A. & Ghorbani, A. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. (2018,1), dx.doi.org/10.5220/0006639801080116

[53] Nagy, R., Nemeth, K., Papp, D. & Buttyan, L. Rootkit Detection on Embedded IoT Devices. *Acta Cybernetica*. (2021,8), https://www.crysys.hu/publications/files/setit/cpaper_bme_NagyB20cscs.pdf

[54] Saleem, D., Anwar, U., Khawar, M. & Naseer, S. Flow-Based Rules Generation for Intrusion Detection System using Machine Learning Approach. (2021,1)

[55] Caida "DDoS Attack 2007" Dataset. 2007, https://www.caida.org/catalog/datasets/ddos-20070804_dataset/ (Accessed on 10.12.2021)

[56] Patil, N., Rama Krishna, C. & Kumar, K. Distributed frameworks for detecting distributed denial of service attacks: A comprehensive review, challenges and future directions. *Concurrency And Computation: Practice And Experience*. **33**, e6197 (2021), https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6197

[57] Shiravi, A., Shiravi, H., Tavallaee, M. & Ghorbani, A. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Computers Security*. **31**, 357-374 (2012), https://www.sciencedirect.com/science/article/pii/S0167404811001672

[58] Intrusion Detection Evaluation Dataset (CIC-IDS2017). 2017, https://www.unb.ca/cic/datasets/ids-2017.html (Accessed on 05.01.2022)

[59] CSE-CIC-IDS2018 on AWS. 2018, https://www.unb.ca/cic/datasets/ids-2018.html (Accessed on 05.01.2022)

[60] Intrusion detection evaluation dataset. 2012, https://www.unb.ca/cic/datasets/ids.html (Accessed on 08.12.2021)

[61] Song, J., Takakura, H., Okabe, Y., Eto, M., Inoue, D. & Nakao, K. Statistical Analysis of Honeypot Data and Building of Kyoto 2006+ Dataset for NIDS Evaluation. *Proceedings Of The First Workshop On Building Analysis Datasets And Gathering Experience Returns For Security*. pp. 29-36 (2011), https://doi.org/10.1145/1978672.1978676

[62] Panigrahi, R. & Borah, S. A detailed analysis of CICIDS2017 dataset for designing Intrusion Detection Systems. *International Journal Of Engineering Technology*. **7** pp. 479-482 (2018,1)

[63] Leevy, J. & Khoshgoftaar, T. A survey and analysis of intrusion detection models based on CSE-CIC-IDS2018 Big Data. *Journal Of Big Data*. **7** (2020,11)

[64] ÃzgÃ¼r, A. & Erdem, H. A review of KDD99 dataset usage in intrusion detection and machine learning between 2010 and 2015. (2016,4)

[65] Liu, M., Xue, Z., Xu, X., Zhong, C. & Chen, J. Host-Based Intrusion Detection System with System Calls: Review and Future Trends. *ACM Comput. Surv.*. **51** (2018,11), https://doi.org/10.1145/3214304

[66] Thakkar, A. & Lohiya, R. A Review of the Advancement in Intrusion Detection Datasets. *Procedia Computer Science*. **167** pp. 636-645 (2020), https://www.sciencedirect.com/science/article/pii/S1877050920307961, International Conference on Computational Intelligence and Data Science

[67] Creech, G. Developing a high-accuracy cross platform Host-Based Intrusion Detection System capable of reliably detecting zero-day attacks. (2014)

[68] Creech, G. & Hu, J. A Semantic Approach to Host-Based Intrusion Detection Systems Using Contiguousand Discontiguous System Call Patterns. *IEEE Transactions On Computers*. **63** pp. 807-819 (2014)

[69] Borisaniya, B. & Patel, D. Evaluation of Modified Vector Space Representation Using ADFA-LD and ADFA-WD Datasets. *Journal Of Information Security*. **6** pp. 250 (2015,7)

[70] Haider, W., Creech, G., Xie, Y. & Hu, J. Windows Based Data Sets for Evaluation of Robustness of Host Based Intrusion Detection Systems (IDS) to Zero-Day and Stealth Attacks. *Future Internet*. **8** (2016), https://www.mdpi.com/1999-5903/8/3/29

[71] Haider, W., Hu, J., Slay, J., Turnbull, B. & Xie, Y. Generating realistic intrusion detection system dataset based on fuzzy qualitative modeling. *Journal Of Network And Computer Applications*. **87** pp. 185-192 (2017), https://www.sciencedirect.com/science/article/pii/S1084804517301273

[72] Kerrisk, M., *perf-trace(1) â Linux manual page*, https://www.man7.org/linux/man-pages/man1/perf-trace.1.html (Accessed on 10.11.2021)

[73] Kerrisk, M., *strace(1) â Linux manual page*, https://www.man7.org/linux/man-pages/man1/strace.1.html (Accessed on 10.11.2021)

[74] Rajendran, S., Calvo-Palomino, R., Fuchs, M., Bergh, B., Cordobes, H., Giustiniano, D., Pollin, S. & Lenders, V. Electrosense: Open and Big Spectrum Data. *IEEE Communications Magazine*. **56**, 210-217 (2018), 10.1109/MCOM.2017.1700200

[75] Bace, R. & Mell, P. NIST Special Publication on Intrusion Detection Systems. (2001)

[76] Xu, L., Zhang, D., Alvarez, M., Morales, J., Ma, X. & Cavazos, J. Dynamic Android Malware Classification Using Graph-Based Representations. *2016 IEEE 3rd International Conference On Cyber Security And Cloud Computing (CSCloud)*. pp. 220-231 (2016)

[77] Tan, K. & Maxion, R. "Why 6?" Defining the operational limits of stide, an anomaly-based intrusion detector. *Proceedings 2002 IEEE Symposium On Security And Privacy*. pp. 188-201 (2002)

[78] Laszka, A., Abbas, W., Sastry, S., Vorobeychik, Y. & Koutsoukos, X. Optimal Thresholds for Intrusion Detection Systems. *Proceedings Of The Symposium And Bootcamp On The Science Of Security*. pp. 72-81 (2016), https://doi.org/10.1145/2898375.2898399

[79] Varghese, S. & Jacob, K. Anomaly Detection Using System Call Sequence Sets. *JSW*. **2** pp. 14-21 (2007,1)

[80] Marteau, P. Sequence Covering for Efficient Host-Based Intrusion Detection. *IEEE Transactions On Information Forensics And Security*. **14**, 994-1006 (2019)

[81] Liu, M., Xue, Z., Xu, X., Zhong, C. & Chen, J. Host-Based Intrusion Detection System with System Calls: Review and Future Trends. *ACM Comput. Surv.*. **51** (2018,11), https://doi.org/10.1145/3214304

[82] Murtaza, S., Khreich, W., Hamou-Lhadj, A. & Couture, M. A host-based anomaly detection approach by representing system calls as states of kernel modules. *2013 IEEE 24th International Symposium On Software Reliability Engineering, ISSRE 2013*. pp. 431-440 (2013,11)

[83] University of New Mexico Computer Science Department, Farris Engineering Center. Computer immune systems - UNMSendmail dataset, 1999, https://www.cs.unm.edu/~immsec/data/live-sendmail.html (Accessed on 19.03.2022)

[84] University of New Mexico Computer Science Department, Farris Engineering Center. Computer immune systems - UNMLoginps dataset, 1999, https://www.cs.unm.edu/~immsec/data/login-ps.html (Accessed on 19.03.2022)

[85] University of New Mexico Computer Science Department, Farris Engineering Center. Computer immune systems - UNMXlock dataset, 1999, https://www.cs.unm.edu/~immsec/data/xlock.html (Accessed on 22.03.2022)

[86] UiPath.com. 2022, https://www.uipath.com/

Abbreviations

ADFA-LD	Australian Defence Force Academy Linux Dataset
ADFA-WD	Australian Defence Force Academy Windows Dataset
ADFA-WD:SAA	Australian Defence Force Academy Windows Dataset:Stealth Attack Addendum
AES	Advanced Encryption Standard
API	Application Programming Interface
CAIDA	Center of Applied Internet Data Analysis
CIC	Canadian Institute for Cybersecurity
CSE	Communications Security Establishment
CC	Command Control
DARPA	Defense Advanced Research Agency
DDoS	Distributed Denial of Service
DL	Deep Learning
DLL	Dynamic Link Libraries
DNS	Domain Name System
DOM	Document Object Model
DoS	Denial of Service
FTP	File Transfer Protocol
GID	Group Identifier
HIDS	Host Based Intrusion Detection System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDS	Intrusion Detection System
IoT	Internet of Things
IMAP	Internet Message Access Protocol
IRC	Internet Relay Chat
ISCX	Information Security Center of Excellence
KDD	Knowledge Discovery and Data Mining
LKM	Loadable Kernel Module
ML	Machine Learning
NIDS	Network Based Intrusion Detection System
NSL	Network Security Laboratory
PAM	Privileged Access Management
PID	Process Identification Number
POP3	Post Office Protocol version 3
RSA	RivestâShamirâAdleman
SDR	Software Defined Radio

SMTP	Simple Mail Transfer Protocol
SSH	Secure Shell Protocol
stide	sequence time-delay embedding
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UNB	University of Brunswick
UNM	University of New Mexico
UPnP	Universal Plug and Play
UUID	Universal Unique Identifier

Glossary

Authentication

Authorization Authorization is the decision whether an entity is allowed to perform a particular action or not, e.g. whether a user is allowed to attach to a network or not.

MIPS Microprocessor without Interlocked Pipelined Stages

List of Figures

List of Tables

Appendix A

Installation Guidelines

A.1 Botnets

A.1.1 Bashlite

Server

First, clone the repository from `https://github.com/hammerzeit/BASHLITE` onto your machine. Open `server.c` and change `MY_MGM_PASS` and `MY_MGM_PORT` to your liking. Save and close the file. Next, compile the `server.c` file with `gcc`. After compiling the file execute the executable with port number 6667 regardless of what your `MY_MGM_PORT` is (the `client.c` is programmed to connect to this port so best to keep it this way) and number of threads as parameters. An example is shown below.

Listing A.1: Initializing the C&C

```
$ gcc server.c -o server
$ ./server 6667 5
```

In another command line, run the following command to connect to the initialized C&C. Here it is important to use the `MY_MGM_PORT` that was specified in `server.c` and the IP address from the device from which the command `./server 8889 5` was started.

Listing A.2: Telnet into C&C

```
$ telnet <ip_address> <MY_MGM_PORT>
```

Next, if prompted, enter the management password to log into the C&C.

Client

For the client also clone the repository onto the clients machine. Change the IP address on line 72 to reflect the IP address of the C&C. Save and exit and execute the compiled file. An example is provided below.

Listing A.3: Connect bot to the C&C

```
$ gcc client.c -o client
$ ./client
```

A.2 Rootkits

A.2.1 Bedevil (bdvl)

As usual, clone the github repository, this time only on the target device. Navigate to setup.py, open the file and change the settings as you see fit. Next, execute the command

```
$ sh etc/depinstall.sh && make
```

This will create the build/ directory that will contain `<PAM_UNAME>.b64` and `bdvl.so.*`. Execute `etc/auto.sh` bash script and provide it with the file location of `<PAM_UNAME>.b64`, like in the example below.

```
$ sh etc/auto.sh <path_to_PAM_UNAME.b64>
```

A.3 Backdoors

A.3.1 theTick

Client

To compile the binary file that needs to be executed on the target device, run the command

```
$ sudo apt-get install libcurl4-openssl-dev
```

After the dependency has been installed, the binary file needs to be created using the following commands:

```
$ cd thetick/src
$ make clean
$ make
```

After this process is done, you will find the executable in the bin folder in the root directory of the repository. Run the executable using the following command

```
$ ./ticksvc ADDR PORT
```

where ADDR refers to the IP Address of the server machine and PORT the port on which the server will be listening on.

Server

On the server we need python to install the necessary dependencies and to execute the malware. Use Python 2.7 preferably to run into less issues. Navigate to the repository root folder and execute the command:

```
$ pip install --upgrade -r requirements.txt
```

This will install all the necessary dependencies. Once this is done the server can be started with the execution of the following command:

```
$ python server.py -b ADDR -p PORT
```

Here the same applies as mentioned above, replace ADDR with the IP Address of the server machine and PORT with the port that the server will be listening on.

A.4 Ransomware

A.4.1 RansomwarePoC

To install this malware, git clone the repository [34], open a command line and navigate to the directory where the newly cloned folder is located. To install the dependencies necessary to run this malware execute:

```
$ pip3 install pycryptodome
```

Once the dependencies have installed the malware is ready to use. The default commands to encrypt and decrypt a device are:

```
$ python3 main.py -e or python3 main.py -e <path>
```

to encrypt and the following to decrypt the device:

```
$ python3 main.py -d or python3 main.py -d <path>
```

Additionally to only encrypt a specific directory on a windows target device specify the directory like for example:

```
$ python3 main.py -p "C:\users\user\desktop\test_ransomware" -e
```

and the same for decrypting the directory:

```
$ python3 main.py -p "C:\users\user\desktop\test_ransomware" -d
```

For encrypting specific directories on Linux devices specify the directory like in the example below:

```
$ python3 main.py -p "/home/user/test_ransomware" -e
```

and likewise to decrypt the directory:

```
$ python3 main.py -p "/home/user/test_ransomware" -d
```

Appendix B

Contents of the ZIP file

This section specifies which files are present in the zip file.

- `BA_Ramon_SolodeZaldivar.pdf` the final version of the thesis as .pdf.
- `BA_Ramon_SolodeZaldivar.zip` containing the latex source code of the thesis.
- `ba_thesis_scripts.zip` the source code of the scripts used in this thesis.
- `midterm.pptx` slides of the midterm presentation held on the 30th of January 2022.

The slides for the final presentation are not included at this stage. The presentation is scheduled for the 20th of April and will be ready for this date.