

POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Detection of anomalous and malicious behavior in IoT devices: a new approach for function verification



Relatori

prof. Luca Ardito
prof. Maurizio Morisio
firma dei relatori

.....
.....

Candidato

Ivan Mineo

firma del candidato

.....

Anno Accademico 2023-2024

† Ai miei zii Pino e Ina

Summary

The current state of Internet of Things (IoT) security is marked by significant vulnerabilities, as the rapid expansion of connected devices has outpaced security measures. IoT devices, ranging from smart home appliances to industrial control systems, are increasingly integrated into everyday life and critical infrastructure, but many of these devices lack robust security protections. Common issues include weak passwords, unpatched software, limited encryption, and insufficient update mechanisms, making them prime targets for cyberattacks.

Malware has played a major role in exploiting these vulnerabilities, with large-scale IoT-focused attacks becoming more frequent and sophisticated. IoT malware often aims to create botnets—large networks of compromised devices used for malicious purposes such as Distributed Denial of Service (DDoS) attacks, data theft, and espionage. Notable examples include the Mirai botnet, which compromised millions of IoT devices and launched crippling DDoS attacks, and more recent variants like Mozi and BotenaGo, which continue to exploit IoT devices.

Insecure IoT devices in both consumer and industrial settings pose a growing risk to privacy, safety, and the stability of global networks. As the number of connected devices continues to grow, improving IoT security through stronger authentication, regular software updates, and better device management is critical to mitigate the impact of malware on the IoT ecosystem.

Acknowledgements

This thesis is dedicated to my family, whose support, love, and belief in me have been the foundation of all my achievements. To my parents, thank you for your endless encouragement and guidance; your sacrifices and wisdom have been my constant source of inspiration.

To my friends, thank you for being my source of laughter, motivation, and understanding. You helped me bearing the challenges along the way, and I'm forever grateful for your presence in my life.

And to my girlfriend, whose love, patience, and encouragement have meant more to me than words can express. Thank you for standing by my side through both the triumphs and trials, always believing in me even when I doubted myself. I dedicate this achievement to you with all my heart.

Contents

List of Figures	8
I Part One	11
1 Introduction	13
1.1 Malware Attacks State-of-the-art	13
1.2 Targeted devices	14
1.3 IoT architectures	16
2 Mirai	19
2.1 Overview	19
2.2 Modus Operandi	20
3 Countermeasures Against Mirai	21
3.1 Malware Analysis	21
3.1.1 Static Analysis	21
3.1.2 Dynamic Analysis	22
3.1.3 Hybrid Analysis	22
3.2 Malware Detection	22
3.2.1 Signature-based	23
3.2.2 Behavioural-based	23
II Part Two	25
4 Exploring Mirai: A Tool for Malware Analysis	27
4.1 Introduction	27
4.2 Overview of Script Objectives	27
4.3 Tools and Libraries Used	28
4.4 Binary Analysis using r2pipe	28
4.4.1 ARM32 System Call Table	28
4.4.2 System call extraction	28
4.4.3 String Analysis	29

4.4.4	Function Analysis and Call Graph Construction	30
4.5	Graph Visualization	30
4.6	NetworkX's functions	31
4.6.1	Closeness_centrality()	31
4.6.2	Betweenness_centrality()	33
4.6.3	Degree_centrality()	35
4.6.4	Pagerank()	36
4.6.5	Louvain_communities()	38
4.7	Output and Serialization	39
III	Conclusion	41
4.8	Conclusion and future developments	43

List of Figures

1.1	Most vulnerable Internet of Things devices worldwide in 2022, by share of IoT vulnerabilities identified	15
1.2	Number of Internet of Things (IoT) connections worldwide from 2022 to 2023, with forecasts from 2024 to 2033	16
2.1	Botnet attack count and family distribution in 2020 H1	19
4.1	Snippet_1	29
4.2	Snippet_2	29
4.3	Snippet_2	30
4.4	Snippet_3	30
4.5	Example of closeness centrality	32
4.6	Example of betweenness centrality	34
4.7	Example of degree centrality	36
4.8	Example of pagerank	37
4.9	Example of Louvain communities algorithm	38

*As long as you live,
keep learning how to live.*
[SENECA]

Part I
Part One

Chapter 1

Introduction

1.1 Malware Attacks State-of-the-art

IoT (Internet of Things) software attacks involve exploiting vulnerabilities in IoT devices, systems, or network software components to compromise security, steal data, disrupt operations, or gain unauthorized access. These attacks target the software layer of IoT devices, which includes their operating systems, applications, firmware, and any software interfaces they use for communication. This thesis will focus on one specific type of attack: malware attacks.

Malware is malicious software designed to exploit or attack devices through their hardware or software. Malware can be classified into several types, including viruses, Trojans, rootkits, and backdoors. In the 1980s, malware primarily consisted of file infectors or boot sector viruses, often spread via floppy disks inserted into the machine. However, as technology advanced and electronic devices became more standardized, malware evolved to target these systems more effectively. IoT, a network of devices connected to the Internet without human intervention, is one of the newer technologies being exploited by malware. Personal computers began to target IoT devices as their capabilities increased. Unlike traditional malware, IoT malware actively scans the Internet for vulnerable devices. It delivers its initial payload, typically a stager script, onto these devices, which then downloads an architecture-specific binary sample. After downloading, the script runs the sample, which communicates with a command-and-control (C2) server. The malware contains scanning modules that enable it to infect additional devices by propagating the sample. Many types of malware initially designed to target personal computers, such as Gamut, Necurs, and Skeyeah, have adapted to attack IoT devices by enhancing their capabilities. [1](#)

Some categories of IoT malware include:

1. **Worm:** This type of IoT malware spreads and propagates automatically across IoT devices. Juniper Threat classifies worms as disruptive malware due to their propagation method. Examples of IoT worms include Mirai, Darlloz, Brickerbot, and Gitpaste-12. [1](#)

2. **Trojan:** A Trojan, also known as a Trojan horse, is a type of IoT malware that appears harmless to users but has hidden malicious functionality. Unlike a virus, a Trojan cannot replicate itself. ProxyM is an IoT Trojan that engages in email spamming and DDoS attacks. [1](#)

3. **Virus:** While the term "virus" is common in computer science, its application to IoT devices can be confusing. IoT viruses behave similarly to traditional computer viruses in that they infect devices through self-replicating malicious code. This makes them difficult to remove and enables complex attacks. Silex, for instance, is an IoT virus that infiltrates a device and renders it permanently unusable—a form of permanent denial-of-service (DoS) attack. [1](#)

4. **Backdoor:** A backdoor is a type of IoT malware that exploits hidden access mechanisms intentionally left by manufacturers. Although these mechanisms may help users meet certain requirements, they often create vulnerabilities. As a result, backdoors are sometimes called the "front doors" of attackers. Tsunami and Bashlite are examples of IoT malware backdoors, sometimes classified as Trojans. [1](#)

5. **Spyware:** IoT spyware allows attackers to monitor or spy on a target's data via an infected device. Examples of IoT spyware include Spybot, Skeeyah, and HNS, which can track users' activities. As IoT device usage increases, so does the number of attacks involving spyware. [1](#)

6. **Ransomware:** IoT ransomware is a type of malware that encrypts data on IoT devices and demands a ransom from the victim in exchange for the decryption key. Once infected, users are unable to access their data until they pay the ransom. Necurs is an example of IoT malware that performs ransomware attacks and other forms of digital extortion. [1](#)

1.2 Targeted devices

In 2022, TVs were the most vulnerable IoT devices, with over half of all identified IoT vulnerabilities affecting them. They were followed by smart plugs and routers, with 13 percent and 9 percent of vulnerabilities, respectively. [2](#) The global number of IoT devices is projected to nearly double from 15.9 billion in 2023 to over 32 billion by 2030, with China expected to lead with around 8 billion consumer devices by 2033.

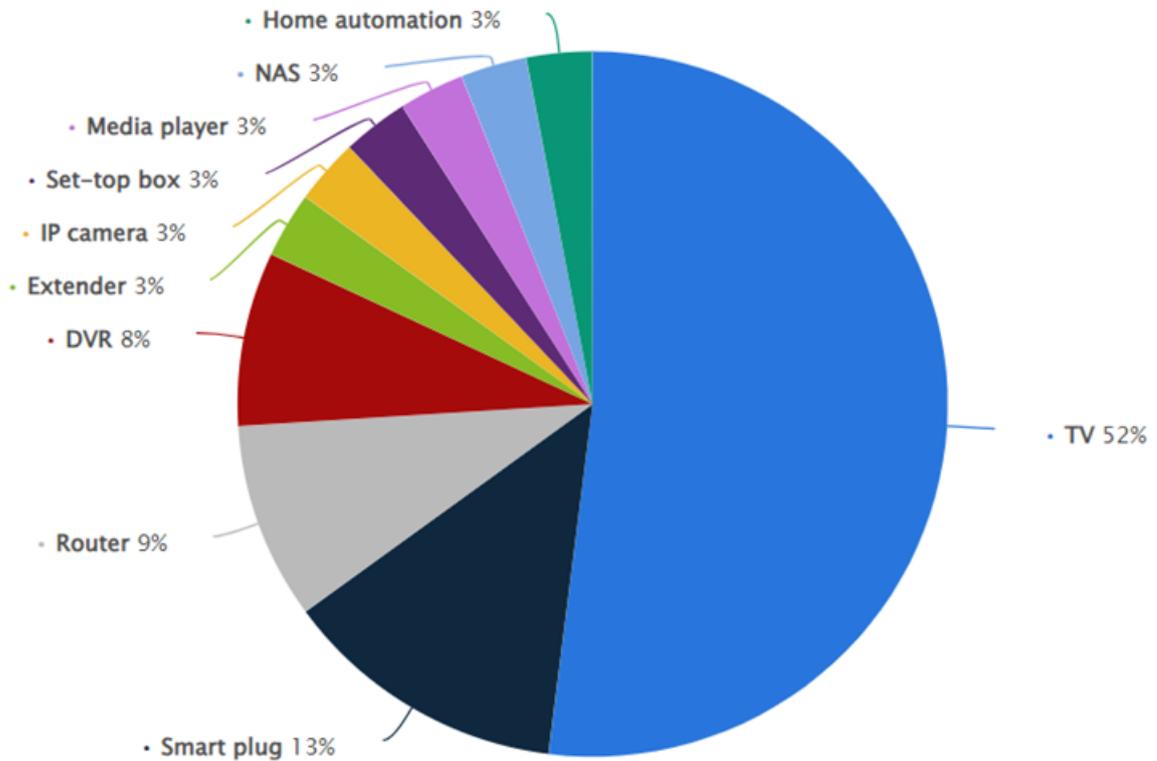


Figure 1.1. Most vulnerable Internet of Things devices worldwide in 2022, by share of IoT vulnerabilities identified

IoT devices are widely used across various industries and consumer markets. In 2023, the consumer segment accounted for approximately 60 percent of all IoT devices, a share that is projected to remain stable over the next decade. Major industry verticals currently utilizing over 100 million connected IoT devices include electricity, gas, water supply, waste management, retail, transportation, and government. By 2033, IoT devices in these sectors are expected to grow to over 8 billion. Key consumer use cases include internet and media devices such as smartphones, which are projected to exceed 17 billion devices by 2033. Other significant use cases with over one billion devices expected by 2033 include autonomous vehicles, IT infrastructure, asset tracking, and smart grids.³

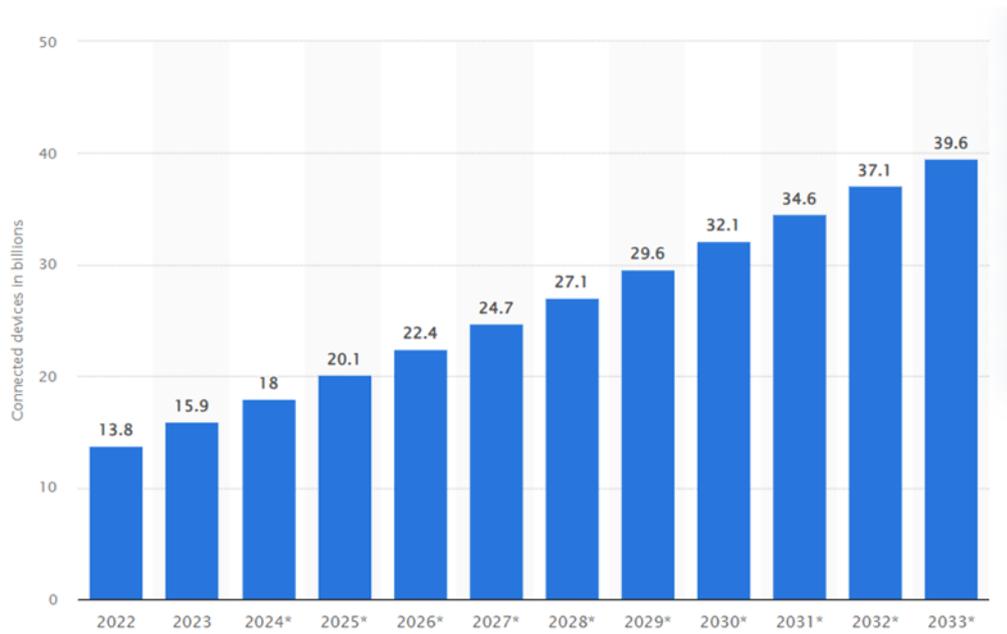


Figure 1.2. Number of Internet of Things (IoT) connections worldwide from 2022 to 2023, with forecasts from 2024 to 2033

1.3 IoT architectures

The Internet of Things (IoT) is revolutionizing how devices connect, communicate, and collaborate. At the heart of IoT lies a diverse range of architectures, each with its unique strengths and applications. These architectures, such as ARM, x86, and MIPS, play a critical role in determining the performance, power efficiency, and capabilities of IoT devices. In this section, we'll discuss about the most widely used.

ARM: ARM-based architectures, rooted in Reduced Instruction Set Computing (RISC), are popular for their energy efficiency, low heat generation, and compact form factor. These processors are well-suited for embedded and low-power applications, such as edge computing, IoT devices, and smart industrial machinery. ARM (Advanced RISC Machine) is commonly used in smartphones, tablets, and other mobile and IoT devices due to its simplified instruction set, which results in low power consumption and small size. ARM processors are also used in servers and data centers where power efficiency is critical. ARM's architecture is licensed to other companies, allowing manufacturers like Qualcomm, Samsung, and Apple to produce their own ARM-based processors.^{4,5}

x86: x86 processors, based on Complex Instruction Set Computing (CISC), are known for their high processing power and compatibility with a broad range of software and hardware. This makes them ideal for PCs, laptops, and demanding industrial applications,

such as control systems and real-time data processing. Developed by Intel in the 1970s and later adopted by AMD, x86 supports operating systems like Windows, Linux, and macOS. While x86 processors are also found in servers and embedded systems, they consume more power and generate more heat compared to ARM processors.^{4,5}

MIPS: MIPS (Microprocessor without Interlocked Pipeline Stages) is a RISC architecture initially developed by MIPS Computer Systems. Known for high performance and low power consumption, MIPS processors have been used in personal computers, servers, mobile devices, and embedded systems. The modular MIPS architecture allows customization for various applications. Although MIPS was widely used in the past, its popularity has declined with the rise of ARM and x86.⁵

PowerPC: PowerPC, developed in the early 1990s by IBM, Apple, and Motorola, is a RISC-based architecture designed for high performance. It was widely used in Apple's Macintosh computers during the 1990s and early 2000s, as well as in some game consoles and consumer electronics. While PowerPC processors are known for their energy efficiency and performance, they have largely been replaced by ARM and x86 in recent years.⁵

Chapter 2

Mirai

2.1 Overview

Mirai is a worm-like malware family that infects IoT devices, integrating them into a botnet for Distributed Denial of Service (DDoS) attacks. Since its initial deployment in 2016-2017, Mirai has infected hundreds of thousands of IoT devices, turning them into remotely controlled bots. Several variants have emerged since then, enhancing the infection process. The malware supports multiple hardware architectures, making it versatile in its targeting. Despite the evolution of these variants, IoT devices, particularly within IoT-based smart grids (SGs), continue to pose significant security risks. [6](#)

Family	Attack Count	Percent
Mirai	64,480	48.965%
Gafgyt	21,923	16.648%
SDBot	21,052	15.986%
YoYo	18,923	14.370%
Dofloo	3220	2.445%
Nitol	1081	0.821%
XorDDoS	708	0.538%
Tianfa DDoS	277	0.210%
Tsunami	21	0.016%
Mayday	2	0.002%

Figure 2.1. Botnet attack count and family distribution in 2020 H1

2.2 Modus Operandi

Mirai initiates its spread through a rapid scanning phase, sending TCP SYN probes to random IPv4 addresses on Telnet ports TCP/23 and TCP/2323, while excluding addresses on a hardcoded IP blacklist. When it identifies a vulnerable device, Mirai attempts a brute-force login using one of ten randomly selected username-password pairs from a pre-configured list of 62 credentials. Upon a successful login, the infected device's IP and credentials are reported to a hardcoded server.

A separate loader program logs into the vulnerable device, identifies its system environment, and downloads and executes the malware. After infection, Mirai conceals its presence by deleting the binary file and renaming its process with a pseudorandom string; however, infections do not persist across reboots. To reinforce control, Mirai terminates other processes associated with TCP/22 or TCP/23, as well as those linked to competing infections. The bot then listens for attack commands from its command-and-control (C2) server while continuing to scan for new victims.

To spread further, Mirai locates IoT devices with open Telnet ports and attempts brute-force logins on those devices. Upon successful infection, these devices send their details back to the C2 server, which instructs them to download the necessary malware binaries. To avoid detection, the malware removes its binary file after execution, running solely in memory. The botmaster can issue attack commands specifying target parameters, with Mirai capable of executing ten different types of DDoS attacks. Since the release of its source code, over 60 variants have emerged, further underscoring the ongoing security risks within the IoT ecosystem. [6](#), [7](#)

Chapter 3

Countermeasures Against Mirai

3.1 Malware Analysis

Malware analysis involves studying malicious files to understand key aspects such as malware behavior, evolution, and target selection. The goal is to help security firms strengthen their defense strategies against malware attacks. Malware analysis techniques are mainly categorized into three types: static, dynamic, and hybrid analysis.

Malware analysis determines how malware functions and answers questions like: How does the malware operate? Which machines and programs are affected? What data is being damaged or stolen? The two primary methods are static and dynamic analysis. Static analysis examines malware without executing the code, while dynamic analysis observes malware behavior during execution. Typically, analysis starts with basic static techniques and progresses to advanced dynamic analysis, often involving reverse engineering and specialized tools to interpret the malware in different formats. [8](#), [9](#)

3.1.1 Static Analysis

Static analysis refers to examining Portable Executable (PE) files without executing them. It helps detect malware by identifying patterns like Windows API calls, string signatures, control flow graphs (CFG), opcode frequency, and byte sequence n-grams.

- **Strings:** Critical for revealing the attacker's intent, as they often contain key semantic information.
- **Control Flow Graph (CFG):** Represents program structure by showing the control flow through nodes (code blocks) and edges (control paths). CFG can capture malware behavior by analyzing the program's structure.
- **Opcodes:** These are the first part of machine code instructions executed by the CPU. Opcode frequency or sequence similarity is used to detect malware.

- N-grams: A sequence of N items (e.g., "MALWARE" → "MAL", "ALW", "LWA") used in malware detection by segmenting strings or sequences of operations.

Additional static features include file size, function length, and networking aspects like TCP/UDP ports and HTTP requests. [8](#)

3.1.2 Dynamic Analysis

Dynamic analysis, also known as behavior analysis, involves running suspicious files in controlled environments (e.g., VMs, emulators) to observe their behavior. This method is effective at detecting both known and unknown malware, including obfuscated and polymorphic types. However, it requires more time and resources than static analysis.

Common techniques include monitoring function calls, analyzing parameters, tracking instructions, and following information flows. API and system calls, along with file system, registry, and network features, are key elements in dynamic analysis.

Malware often uses anti-virtual machine and anti-emulator techniques to evade detection by behaving normally when such environments are detected. Emulators, debuggers, simulators, and virtual machines (VMs) are all used in dynamic analysis, but sophisticated malware can sometimes detect these controlled environments and attempt to avoid analysis. [8](#)

3.1.3 Hybrid Analysis

Hybrid analysis combines static and dynamic methods, providing the advantages of both. Static analysis is fast, cheap, and safe but can be evaded by obfuscation. Dynamic analysis is more reliable and can detect variants and unknown malware, though it is resource-intensive. By merging the two approaches, hybrid analysis improves malware detection accuracy. [8](#)

3.2 Malware Detection

Malware detection methods can be categorized from different perspectives. This section covers two primary methods: Signature-based and Heuristic-based detection.

In the early days, signature-based detection was widely used. It is effective and fast for identifying known malware but struggles with zero-day malware. Over time, techniques such as behavior-based, heuristic-based, and model-checking-based detection have emerged, along with newer methods like deep learning-based, cloud-based, mobile device-based, and IoT-based detection.

Although behavior-based and heuristic approaches can detect many malware types, including new ones, they do not cover all malware. The challenge remains to develop a method that effectively detects more complex and unknown malware. [8, 9](#)

3.2.1 Signature-based

Most antivirus software relies on signature-based detection. This method extracts unique signatures—sequences of bytes or file hashes—from malware files to identify and detect similar malware. Signature-based detection has a low false positive rate but is vulnerable to evasion by attackers who can easily modify the malware signature.

Signature-based detection is fast and effective for known malware but fails with new malware due to obfuscation techniques like dead code insertion and instruction substitution. Creating an effective signature requires:

- Compactness to represent multiple malware with one signature,
- Efficient automatic generation mechanisms,
- Integration of data mining and machine learning techniques,
- Resistance to packing and obfuscation methods.

Despite its success with known malware, signature-based detection is insufficient for modern, complex threats like polymorphic malware. [8](#), [9](#)

3.2.2 Behavioural-based

Behavior-based detection analyzes a file’s runtime activities to identify malware. In a learning phase, patterns are extracted from the file, and in a testing phase, the file is classified as malicious or legitimate based on those patterns. This method can detect unknown malware and those using obfuscation techniques. However, it has a higher false positive rate and can be time-consuming.

Heuristic-based approaches, often relying on data mining techniques like Support Vector Machines, Naïve Bayes, Decision Trees, and Random Forests, help detect malware by analyzing behavior patterns.

Behavioral detection generally involves three steps:

1. Identify behaviors (using data mining),
2. Extract features from behaviors,
3. Classify the file using machine learning.

While behavior-based methods are effective in detecting new malware families, challenges remain, such as handling large numbers of features, similarity detection, and the inability of some malware to run in virtual environments. Advances in machine learning and data mining techniques are playing a significant role in improving malware detection by better interpreting features. [8](#), [9](#)

Part II
Part Two

Chapter 4

Exploring Mirai: A Tool for Malware Analysis

4.1 Introduction

The Python script provided operates at the intersection of binary analysis, system call tracing, and graphical representations of program flows. It leverages several tools and libraries to achieve these tasks, including `r2pipe`, `networkx`, `pygraphviz`, and `matplotlib`. The purpose of this script is to analyze ARM32 binaries, extract specific system calls, and represent relationships between functions and strings within the binary in the form of a directed graph.

This chapter explains the major components of the script, the libraries it uses, and how they function together to produce meaningful insights from ARM binaries.

– MAY ADD GITHUB REPOSITORY LINK

4.2 Overview of Script Objectives

The primary goals of the script are:

1. **Binary Disassembly and Analysis:** Using `r2pipe` to perform an in-depth analysis of an ARM binary and extract relevant disassembly information.
2. **System Call Identification:** Parsing system calls specific to ARM32 and associating them with corresponding function addresses.
3. **String Extraction and Mapping:** Identifying and mapping strings within the binary to functions, aiding in reverse engineering and vulnerability analysis.
4. **Graph Construction and Network Analysis:** Utilizing `networkx` and `pygraphviz` to construct and analyze a directed graph representing function dependencies and their associations, followed by centrality and community detection analyses.

4.3 Tools and Libraries Used

The script uses several specialized libraries for binary analysis and graph generation, including:

- **r2pipe**: A Python binding for radare2, a popular framework for reverse engineering and analyzing binaries. It allows this script to interact with the binary, extract relevant data (such as system calls), and perform automated tasks like disassembly and string analysis.
- **pygraphviz**: A Python interface to the Graphviz graph layout and visualization software, used for creating and visualizing the relationships between functions and strings.
- **networkx**: A library for creating, manipulating, and analyzing complex networks of nodes and edges. This is crucial for building the call graphs and visualizing the binary's flow.
- **matplotlib**: A popular Python plotting library used to visualize graphs generated from networkx.

These libraries provide the foundation for automated binary dissection, data structure creation, and visualization.

4.4 Binary Analysis using r2pipe

At the core of this script is the use of r2pipe, which connects the script to radare2. Radare2 is a powerful reverse-engineering framework capable of disassembling and analyzing binary executables. The script opens a binary file specified via command-line argument using `r2pipe.open(binary_path)` and then executes the command `aaa` to perform analysis of the binary, which includes detecting functions, system calls, and other important structures.

4.4.1 ARM32 System Call Table

A dictionary named `ARM32_SYSCALLS` defines a mapping between ARM32 system call numbers and their corresponding system call names. This is essential for decoding system calls found within the binary, converting them from their numeric forms into human-readable names.

```
ARM32_SYSCALLS = 0: "restart_syscall", 1: "exit", ...
```

This dictionary includes all the system calls in the ARM's ISA. As the script parses the binary, it uses this dictionary to map numerical syscalls to their readable names.

4.4.2 System call extraction

The script parses the binary to identify system calls. It uses regular expressions to detect specific patterns in the disassembled code, particularly focusing on supervisor calls (`svc`)

in the ARM architecture (`r.cmdj('/atj swi')`) is an equivalent command in Radare2 to find svc instructions). The script filters relevant system calls into two lists (`svcList1` and `svcList2`) based on certain criteria such as hexadecimal prefixes and code offsets.

```
svcs = r.cmdj('/ad/j svc [0-9a-fA-F]+')
for s in svcs:
    s1 = s['code'].split(" ")
    s2 = str(s1[1])
    if s2.startswith("0x9"):
        svcList1.append(s)
    if len(s2) <= 2:
        svcList2.append(s)
```

Figure 4.1. Snippet_1

From this information, the script creates instances of a `Syscall` class, each holding details like the hexadecimal representation, offset in the binary, and the decoded system call name from the `ARM32_SYSCALLS` dictionary.

4.4.3 String Analysis

The script also performs string extraction using the command `izj` within radare2. The strings extracted from the binary are stored in instances of the `StrC` class ("StringClass"), along with references to the functions where those strings are used (`axtj` command). This data is crucial for later visualizing which strings are linked to which functions.

```
strings = r.cmdj('izj')
for st in strings:
    refs1 = r.cmdj(f"axtj {st['vaddr']}")
    if refs1:
        strObj = StrC()
        strObj.text = st['string']
        strObj.funAddr = [rf['fcn_addr'] for rf in refs1 if rf.get('fcn_addr')]
        allStrings.append(strObj)
```

Figure 4.2. Snippet_2

4.4.4 Function Analysis and Call Graph Construction

To build a call graph, the script gathers information about functions within the binary. This is done using radare2's graph analysis commands (`agCj`, `agfj`). For each function found, the script identifies any associated system calls, strings used, and imports. This data is stored in instances of the `Funx` class ("FunctionClass").

```
agCj = r.cmdj('agCj')
for f in agCj:
    agfj = r.cmdj(f"agfj @ {f['name']}")
    funObj = Funx()
    funObj.name = agfj[0]['name']
    funObj.offset = agfj[0]['offset']
    # Collect system calls, imports, and strings
```

Figure 4.3. Snippet_2

4.5 Graph Visualization

With the extracted functions, system calls, and strings, the script uses `NetworkX` to build a directed graph (`nx.DiGraph()`), where each node represents a function or string, and edges represent the relationships between them.

The nodes are color-coded based on their type:

- **Functions:** Blue or Cyan, depending on the presence or not of system calls
- **Strings:** Red

The graph's layout is computed using the `spring_layout` algorithm, and `matplotlib` is used to draw the graph with labels indicating the node content (function names, system calls, etc.).

```
G = nx.DiGraph()
for d in data:
    if len(d["content"]) > 0:
        G.add_node(d["name"], content=d["content"], color='b')
    else:
        G.add_node(d["name"], content="/", color='c')
nx.draw_networkx(G, pos, with_labels=True, labels=nx.get_node_attributes(G, 'content'))
```

Figure 4.4. Snippet_3

The script saves the call graph and associated information to a JSON file and optionally as a .dot or .gml file for further analysis.

4.6 NetworkX's functions

All the measures collected from the binaries using the NetworkX functions are the following:

- **Centrality Measures:** Betweenness, closeness, and degree centralities are computed to identify key functions within the graph. These metrics help determine the importance and influence of functions in the binary.
- **PageRank Analysis:** PageRank scores are calculated to rank functions based on their interconnectedness, providing insights into critical components.
- **Community Detection:** The script uses the Louvain method for community detection to identify clusters of related functions, which may indicate modular or cohesive functionality.

For all of these, a dedicated subgraph of the full graph is being created (removing just the string nodes).

4.6.1 Closeness centrality()

Closeness centrality measures how close a node is to all other nodes in a network. It is based on the average shortest path, or "geodesic path," from a given node to all others. For each node, the mean distance to other nodes is calculated, with nodes that are closer to others on average having lower values. These nodes typically have better access to information and can influence other nodes more directly.

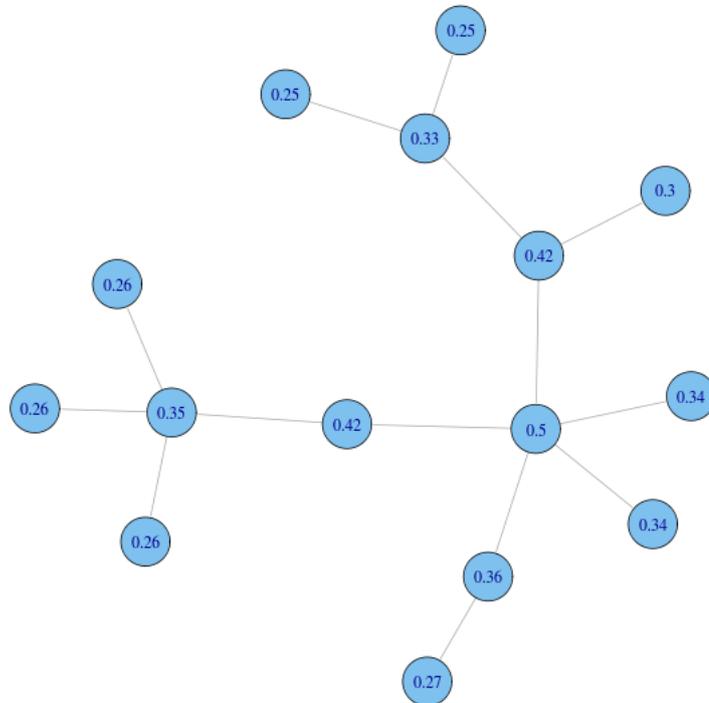


Figure 4.5. Example of closeness centrality

To make this measure more intuitive, its inverse is commonly used, which gives higher values to more central nodes. This is called closeness centrality, and it is calculated as the reciprocal of the average shortest path distance for a node. Essentially, nodes with higher closeness centrality can reach other nodes more quickly.

Unlike other centrality measures like degree or eigenvector centrality, closeness centrality reflects how well-connected a node is based on distance rather than the number of direct connections. For example, a node might have low degree but still be highly central if it is connected to a node that is well-positioned in the network.

Closeness centrality has a few limitations. In disconnected networks, the distance between unreachable nodes is considered infinite, giving a closeness centrality score of zero. To address this, analysts often calculate closeness centrality only for nodes within the largest connected component or assign a large distance for unreachable pairs to make calculations feasible.

One challenge is that the range of closeness centrality values tends to be small, making it

difficult to distinguish between highly central and less central nodes. Even minor changes in the network structure can significantly affect the closeness ranking of nodes. [10](#)

-fix- The closeness centrality function in NetworkX can be used to analyze malware functions in a system call or behavioral graph by measuring how "close" a node (representing a malware function) is to all other nodes in the graph. It quantifies the importance of a node based on the average length of the shortest paths between that node and all other nodes. For malware analysis, it can highlight functions that have a central role in the malware's overall execution flow.

Functions with high closeness centrality are good candidates for security focus, as interrupting them could disrupt the entire malware's operations.

Using NetworkX's `closeness_centrality()` allows you to identify which malware functions are central to the overall execution of malware. This can be particularly useful in prioritizing remediation strategies, focusing on disrupting the most crucial components of the malware's behavior. By targeting functions with high closeness centrality, analysts can potentially interrupt the malware's core operations and limit its spread or impact.

4.6.2 `Betweenness_centrality()`

Betweenness centrality measures how often a node appears on the shortest paths between other nodes in a network. A node with high betweenness centrality has significant control over the flow of information because many paths between other nodes pass through it. Such nodes are crucial for communication, and their removal would greatly disrupt the network.

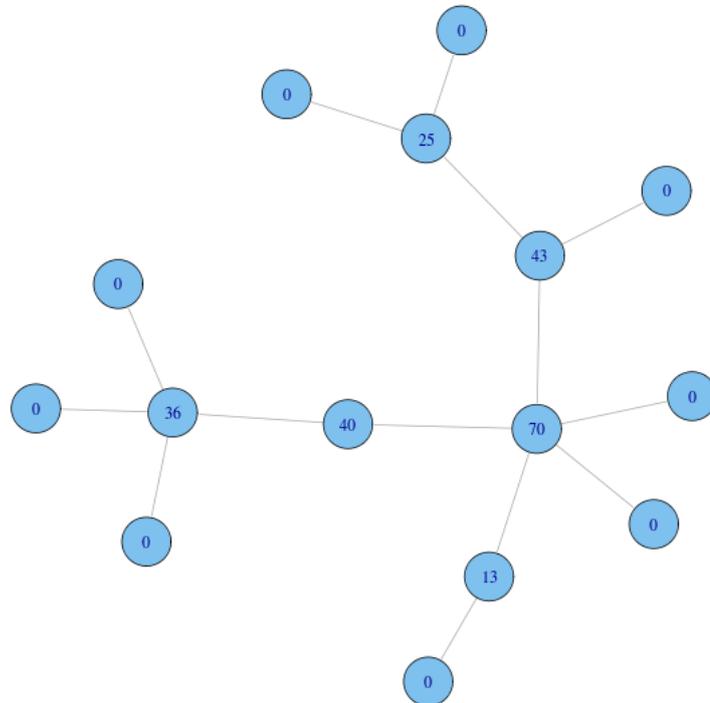


Figure 4.6. Example of betweenness centrality

Mathematically, betweenness centrality is calculated by counting the number of shortest paths between pairs of nodes that pass through a particular node, divided by the total number of shortest paths between those pairs. If no shortest paths pass through a node, its betweenness contribution is zero. This centrality differs from others like degree or closeness centrality. For example, a node might have a low number of direct connections (low degree) or be far from others (low closeness), but if it acts as a bridge between different groups of nodes, it can still have high betweenness. Such nodes are often referred to as "brokers."

The highest possible betweenness occurs in a star network, where the central node connects all others. On the other hand, leaf nodes, which connect to the network with only one edge, have the lowest betweenness.

Unlike closeness centrality, betweenness centrality values usually span a wide range, making it easier to distinguish the most influential nodes in a network. In practice, calculating betweenness centrality involves finding the shortest paths between all node pairs, which

can be computationally intensive, but is manageable in sparse networks where the number of connections is relatively low compared to the number of nodes. [11](#)

-fix- Using NetworkX's betweenness centrality function for analyzing malware functions in a system call or behavior graph can be extremely helpful in identifying key points of control or influence within the malware's execution flow. Betweenness centrality measures how often a node (malware function) appears on the shortest paths between other pairs of nodes. In malware analysis, this can help detect functions that act as bottlenecks or essential intermediaries in the malware's behavior.

Functions that have high betweenness centrality are often critical "intermediaries" between different stages of malware operation.

In malware, identifying bottlenecks in execution paths can help pinpoint critical points for breaking the malware's functionality. By neutralizing or isolating functions with high betweenness centrality, analysts may be able to disrupt the malware's core operations.

Using `betweenness_centrality()` in malware analysis can help you identify critical "choke-points" in the malware's flow. These are functions that serve as key intermediaries and are likely to have a large influence on the malware's ability to execute or spread. By disrupting these high-betweenness functions, security analysts can effectively hinder the malware's operational capabilities, making it a valuable tool in prioritizing mitigation efforts.

4.6.3 Degree_centrality()

Degree centrality computes the degree centrality for nodes. The degree centrality for a node v is the fraction of nodes it is connected to. [12](#)

The degree of centrality can be calculated for both directed and undirected networks and is based on the connection of nodes with numerous edges. In controlled networks, the node's centrality is the total number of incoming and outgoing links; in undirected networks, it is the sum of links. The centrality degree measures the node's direct influence on the local network. This is because nodes with the most incredible power of centrality usually have a more direct impact on the nodes they are connected. The degree of centrality can also be used to identify influential nodes that are part of a shorter distance between two nodes that connect a different network component. [13](#)

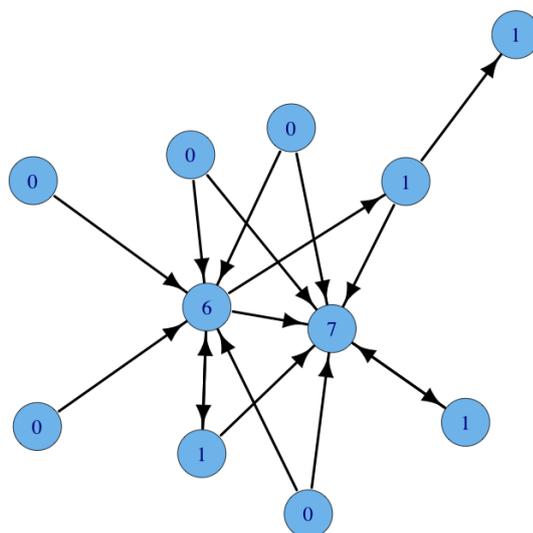


Figure 4.7. Example of degree centrality

-fix- NetworkX's degree centrality function is used to compute the centrality of nodes based on the number of connections (or edges) they have in the graph. In malware function analysis, this can be a powerful tool to identify the most influential or critical functions in the malware's execution flow, potentially highlighting which functions are heavily involved in the malware's operation.

In some cases, high centrality functions can reveal the communication structure of the malware, such as how often certain functions communicate with others.

This function is a useful tool in malware function analysis to identify the most central and connected functions in the malware's execution graph. Functions with high degree centrality likely play critical roles in the malware's operation, making them prime targets for reverse engineering and mitigation efforts.

4.6.4 Pagerank()

PageRank is an algorithm originally developed by Google to measure the importance of web pages. It was named after one of Google's founders, Larry Page, and is based on the idea that pages with more incoming links, especially from other important pages, are considered more valuable. The algorithm counts both the number and quality of links to a page to estimate its importance.

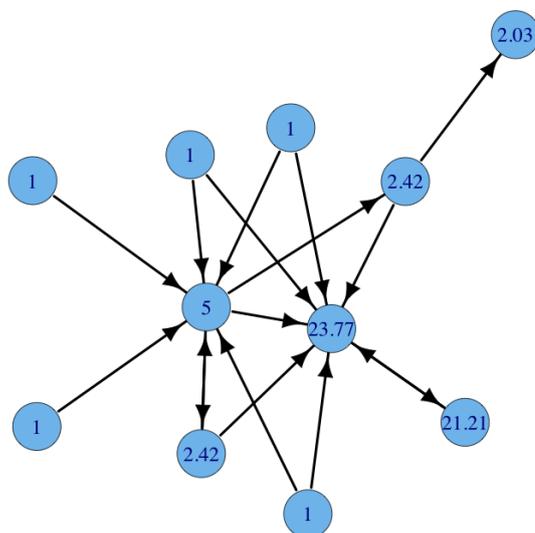


Figure 4.8. Example of pagerank

In simple terms, PageRank works by simulating a random user clicking on links across the web. Each time the user lands on a page, the importance of that page increases based on the PageRank values of the pages that linked to it. The more high-quality links a page has, the more PageRank it accumulates.

Initially, each page is given an equal score, but the algorithm refines these values over several iterations. During each iteration, a page shares its PageRank equally among all the pages it links to. For example, if a page has two outbound links, it will divide its score between those two. Over time, pages that receive links from many other important pages see their PageRank grow, while those with fewer or less significant links see their score shrink.

The formula for PageRank includes a damping factor, which simulates the likelihood that a user will randomly stop clicking and jump to another page. This prevents the system from overly inflating the value of certain pages and ensures the algorithm converges to a stable solution. [14](#)

-fix- NetworkX's `pagerank()` function can be a powerful tool in analyzing malware functions in a graph of system calls. It can be adapted to rank malware functions by their significance or influence in the execution flow.

In malware like botnets or ransomware, certain functions are central. Using PageRank, you can identify these critical functions that serve as important points in the malware's operation.

This can provide deep insights into the critical functions or system calls within malware,

helping analysts understand the structure and prioritize efforts for mitigation. By highlighting the most influential points, PageRank enables a focused approach to malware analysis and remediation.

4.6.5 Louvain_communities()

The Louvain algorithm is a method used to detect communities in large networks by maximizing a modularity score, which measures how well the nodes are grouped into communities compared to a random arrangement. In essence, it seeks to find clusters of nodes that are more densely connected internally than with the rest of the network.

The Louvain algorithm operates hierarchically and consists of two main iterative phases:

1. **Local Movement of Nodes:** Initially, every node in the network is assigned to its own community. The algorithm then evaluates if moving a node to a neighboring community would increase the modularity. If moving the node results in a positive gain in modularity, the node is reassigned to that community. This process continues for all nodes until no further improvements can be made, achieving a local maximum of modularity.
2. **Network Aggregation:** In this phase, the network is condensed by treating each community identified in the first phase as a single node. The links between these new nodes are given weights based on the sum of the weights of the edges between nodes in the corresponding original communities. Once the network is aggregated, the first phase is reapplied to this new, simplified network.

These steps are repeated iteratively until no further improvements in modularity are possible. This results in a hierarchical structure, revealing communities within communities. The Louvain method is widely used because it is relatively fast and easy to implement, but it requires significant memory to store the network data. [15](#), [16](#), [17](#)

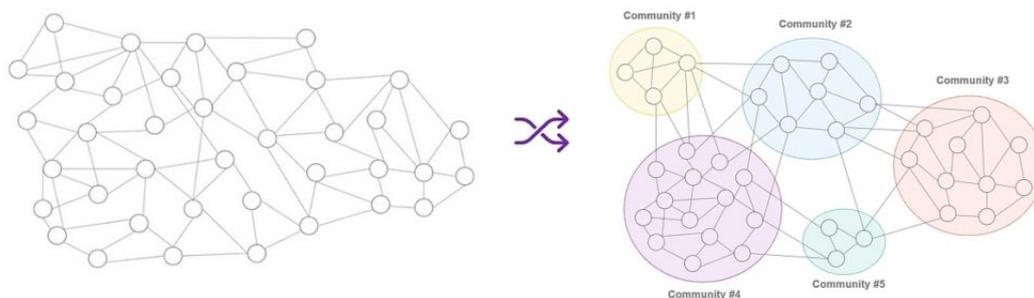


Figure 4.9. Example of Louvain communities algorithm

-fix-The `louvain_communities()` function from NetworkX can be used for identifying communities within a graph, based on the Louvain method for community detection. In the context of analyzing malware syscalls, it is possible to use this function to gain

insights into groups of related system calls that may represent different functional modules or behaviors of the malware. With the `louvain_communities()` function, we can identify clusters or communities within this syscall graph. Each community might represent a set of syscalls that frequently occur together or are closely related in terms of their functionality. For instance:

- One community might represent syscalls related to file operations.
- Another might be associated with network communications.
- Yet another could be linked to process manipulation.

By analyzing these communities, we can gain insights into the modular structure of the malware. This can help in understanding:

- **Functional Segmentation:** Different functional areas of the malware, such as data exfiltration, persistence mechanisms, or evasion techniques.
- **Detection and Mitigation:** Focusing on critical syscall groups for designing more targeted detection rules or countermeasures.
- **Prioritization for Further Analysis:** Syscall communities that are more central or densely connected might be more critical to the malware's operation. Analysts can prioritize these areas for deeper static or dynamic analysis.

4.7 Output and Serialization

The results of the analysis, including centrality scores and community structures, are written to text files. The complete function call graph is serialized into a JSON format for further inspection or visualization.

Part III

Conclusion

4.8 Conclusion and future developments

The Python script offers a comprehensive toolkit for malware analysis in the ARM32 architecture. By automating the extraction of system calls, functions, and strings, and by mapping their interactions in a graph, it enables cybersecurity professionals to analyze malware quickly and effectively. The visual representation of the binary's behavior, coupled with NetworkX most useful functions, provides deep insights into the malware's operation, making this script a powerful tool in identifying, understanding, and mitigating malicious threats.

This script has significant potential for future developments that could enhance its capabilities for malware analysis. As the cyberattacks landscape evolves, and malware becomes more sophisticated, there are several areas where the script can be improved to provide deeper insights and more effective analysis of malware binaries. This section will explore possible future developments, focusing on improving automation, integration with advanced techniques, and expanding the scope of analysis.

- **Machine Learning Integration for Automated Malware Classification:**
One potential enhancement is integrating machine learning (ML) techniques to classify malware based on the patterns detected in the binary analysis. By training models on large datasets of known malware samples, the script could automatically classify new malware into known families or identify novel patterns that might indicate previously unseen threats. Machine learning models could be trained to recognize malicious patterns based on system calls, control flow graphs, and function behavior.
- **Expanding Platform Support Beyond ARM32:**
Although the current script is tailored for ARM32 binaries, expanding its capabilities to support a wider range of architectures and operating systems would significantly broaden its use cases. With the rise of malware targeting different platforms, such as x86, x64, MIPS, and RISC-V, enhancing cross-platform compatibility would make the script more versatile for analyzing malware across a wider range of environments. This would allow the script to be used by a broader range of analysts, and it would help detect malware that targets systems beyond ARM32-based IoT and embedded devices.
- **Improved Visualization and Reporting:**
The current graph-based visualization of function relationships and system calls is a powerful tool for malware analysis, but it could be enhanced with more detailed and customizable visualizations, along with automated reporting features. Providing richer insights through advanced graph analytics and improving the user experience would help make the script more accessible to analysts of all skill levels. Better visualization would make it easier to interpret complex malware behaviors, and automated reporting would streamline the process of documenting findings for incident response or forensic investigations.

- **Enhancement of Dynamic Analysis Capabilities:**

Currently, the script focuses primarily on static analysis of binaries, which examines the code without executing it. Adding dynamic analysis capabilities—running the malware in a controlled environment or sandbox—would allow the script to observe real-time behavior, providing insights that static analysis alone may miss. Dynamic analysis would provide a complementary view of the malware, allowing the script to detect behaviors that are only triggered under specific conditions or during execution.

The Python script has considerable potential for future developments that would make it an even more powerful tool for malware analysis. Integrating machine learning for automated classification, enhancing dynamic analysis, expanding platform support, improving visualizations and collaborating with threat intelligence platforms are just some of the areas where this tool can evolve. As malware becomes more sophisticated and diversified across platforms, enhancing the script's capabilities will help security analysts stay ahead of emerging threats, making it an indispensable asset for cybersecurity defense in the years to come.

Bibliography

1. URL <https://www.sciencedirect.com/science/article/pii/S2949715923000793#sec7>.
10. URL <https://www.sci.unich.it/~francesc/teaching/network/closeness.html>.
11. URL <https://www.sci.unich.it/~francesc/teaching/network/betweeness.html>.
12. URL <https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms centrality.degree centrality.html#networkx.algorithms centrality.degree centrality>.
13. URL <https://www.sciencedirect.com/book/9780443190964/emotional-ai-and-human-ai-interactions-in-social-networking>.
14. URL <https://www.geeksforgeeks.org/page-rank-algorithm-implementation/>.
15. URL <https://towardsdatascience.com/community-detection-algorithms-9bd8951e7dae>.
16. URL <https://neo4j.com/docs/graph-data-science/current/algorithms/loouvain/>.
17. URL https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.community.loouvain.loouvain_communities.html#networkx.algorithms.community.loouvain.loouvain_communities.
2. URL <https://www.statista.com/statistics/1406530/most-vulnerable-iot-devices-by-share-of-vulnerabilities-worldwide/>.
3. URL <https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/>.
4. URL <https://www.neousys-tech.com/edge-ai-computing/knowledge/x86-and-arm-based-for-industrial-computing.html>.
5. URL <https://www.windriver.com/solutions/learning/leading-processor-architectures>.

BIBLIOGRAPHY

6. URL <https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-antonakakis.pdf>.
7. URL https://www.researchgate.net/publication/360489799_Using_Delphi_and_System_Dynamics_to_Study_the_Cybersecurity_of_the_IoT-Based_Smart_Grids.
8. URL <https://core.ac.uk/download/pdf/325990564.pdf>.
9. URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8949524>.