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

IoT security is currently riddled with remarkable vulnerabilities, where sudden growth in the connected devices outpaced security. From smart home appliances to industrial control systems, these IoT devices increasingly make up much of everyday living and critical infrastructure, but most massively lacks robust security protections. Weak passwords, unpatched software, limited encryption, and poor mechanisms for updating are just a few of the common issues that make them prime targets for cyberattacks.

IoT-focused attacks have increased both in frequency and complexity, and malware has played an important role in the exploitation of IoT device vulnerabilities. Malware in IoT mostly tries to create an army of bots: large-scale networks of compromised devices intended for malicious ends such as DDoS attacks, data theft, and espionage. Notable examples are the Mirai botnet-which compromised millions of IoT devices before using them to run a DDoS attack on several well-known websites-and the BrickerBot malware that bricks vulnerable IoT devices by corrupting their flash storage.

The crippling DDoS attacks, including lately variants such as Mozi and BotenaGo, continue to compromise IoT devices. Insecure IoT devices in consumer and industrial contexts pose a growing threat to privacy and safety that resonates through global network stability. To have better IoT security, more connected devices need an improved authentication process. Software updating and device management would be other steps to decrease the aftermaths of malware on IoT ecosystems.

# 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

*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 result in using vulnerabilities in IoT devices, systems, or network software components to compromise security, steal data, disrupt operations, or gain unauthorized access. Such attacks focus on the software layer of IoT equipment, such as operating systems, applications and firmware, of any software interfaces they use to communicate. This thesis will focus on one particular attack type: 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 was almost exclusively file infectors or boot sector viruses, often spread via floppy disks inserted into the computer. However, as technology advanced and electronic devices became standardized, malware began to evolve to target these systems better. IoT, a network of devices connected to the Internet without human intervention, is one of the newer technologies being exploited by malware. As personal computers' powers grew, so did their targets to IoT devices. Unlike traditional malware, IoT malware is scanning the Internet actively to find vulnerable devices.
It downloads its first payload, commonly a stager script, to these machines, which in turn 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, which enables it to propagate to other devices through the sample. Many types of malware initially designed to target personal computers, such as Gamut, Necurs, and Skeeyah, have been repurposed to target IoT devices by beefing up 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 refers to malware that encrypts data on the IoT. devices and then asks for a ransom from the targeted victim in return for the decrypting key. Once infected, users are unable to access their data until they have paid the ransom. Necurs is one such example of IoT malware, which carries out ransomware attacks among 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.

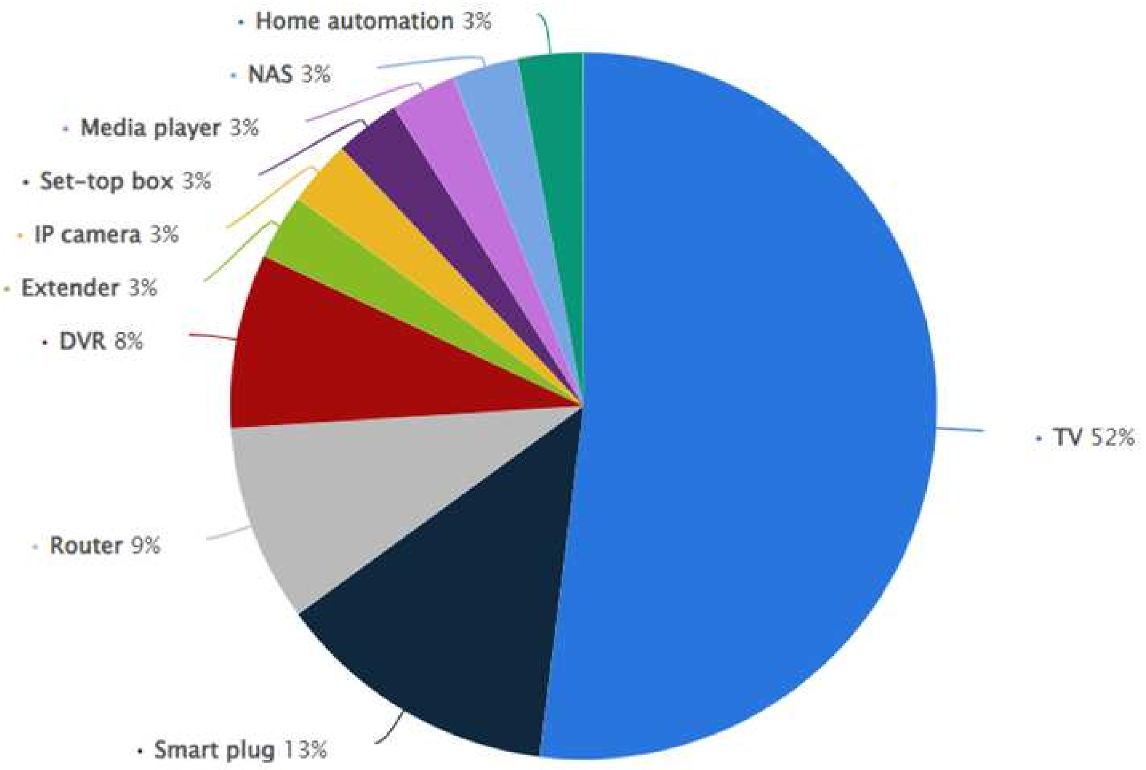


Figure 1.1. Most vulnerable Internet of Things devices worldwide in 2022, by share of IoT vulnerabilities identified

IoT devices have a widespread application in multiple industries and consumer markets. In 2023, The consumer segment accounted for about 60 percent of all IoT devices, a share that is projected to remain flat over the coming decade. Top industry segments currently using 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 projected to reach over 8 billion. Critical consumer use cases are the internet and media devices such as smartphones, which are expected to eclipse 17 billion devices by 2033. Other top use cases with over one billion devices forecasted by 2033 include autonomous cars, IT infrastructure, tracking of the assets, and smart grids.[3]

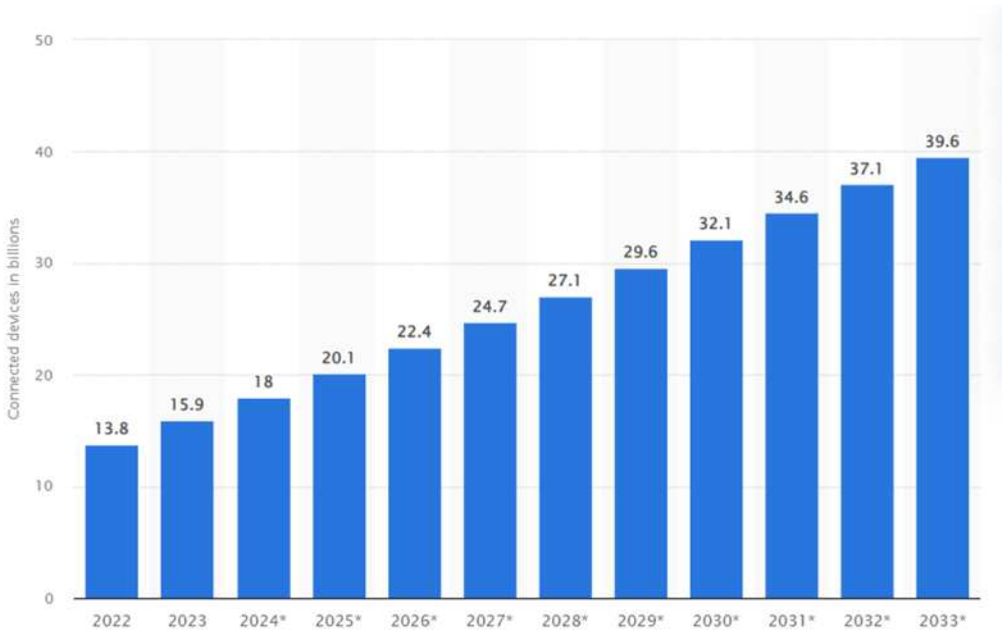


Figure 1.2. Number of Internet of Things (IoT) connections worldwide from 2022 to 2023, with forecasts from 2024 to 2033

## 1.3 IoT architectures

The IoT is fundamentally changing the way devices will connect, communicate, and interact with one another. At the core of IoT is a bevy of architectures, each with its unique strengths and application. These architectures serve to determine the performance, efficiency, and capability of IoT devices. This section discusses the most widely used architectures: ARM, x86, and MIPS.

**ARM**: ARM architectures are RISC-based; this processor has gained much popularity due to its power efficiency, low heating, and space-saving format. These processors are suitable for embedded and low-power applications: edge computing, IoT devices, and smart industrial machinery. Lastly, due to their structure, ARM processors have increasingly become common in recent times in smartphones and tablets among other mobile and IoT devices. It uses low power and is hence smaller in size by having a reduced instruction set. ARM processors are also installed in servers and data centers where power efficiency is crucial. ARM licenses its architecture to other companies; that is how manufacturers like Qualcomm, Samsung, and Apple make their ARM-based processors. [4,5]

**x86**: The x86 processor, based on Complex Instruction Set Computing, or CISC, is known for their high processing power, compatibility with a vast range of software and hardware, they find their best fit in PCs, laptops, and also in very demanding industrial purposes, such as control systems or real-time data processing. Initially, x86 was created by Intel in the 1970s, and then AMD followed suit. x86 supports operating systems like Windows, Linux, and though existing in both server and embedded systems, x86 processors also have a higher power consumption and dissipation than their ARM counterpart. [4,5]

**MIPS**: The MIPS, initially developed at the MIPS Computer Systems, is a RISC architecture. The fact that it showed high performance with low power dissipation has seen its application expanded from personal computers and servers to even mobile and embedded systems. The architecture of the MIPS is modular, which customisation to suit a variety of needs. Once extremely ubiquitous, MIPS has, in recent times, fallen by the wayside due to the increase prominence of ARM and x86. [5]

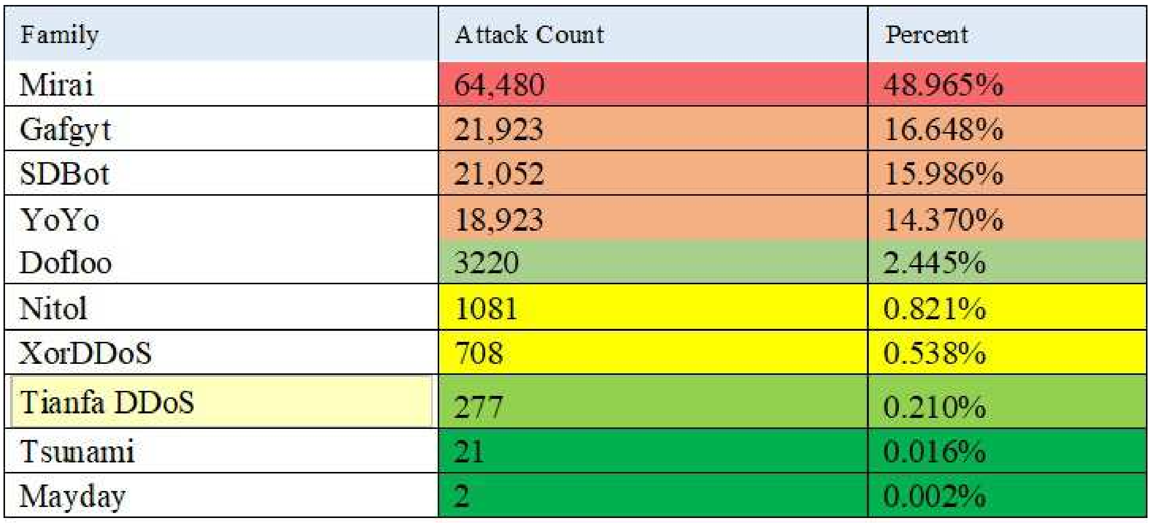
**PowerPC**: The PowerPC architecture, created by IBM, Apple, and Motorola during the early 1990s, was targeted at high performance and based on RISC. With the exception of a few game consoles and consumer electronics, it has led an extensive run at Apple's Macintosh between the 1990s and the early 2000s. While the PowerPC processors have impressively performed with energy efficiency, recently they have been outcompeted by both the ARM and x86. [5]

# Chapter 2

# Mirai

## 2.1 Overview

Mirai is a worm-like malware family that infects IoT devices to form a botnet for conducting DDoS attacks. Since its initial deployment in 2016-2017, hundreds of thousands of IoT devices have been infected by Mirai and turned into remotely controlled bots. Since then, several variants have emerged that enhanced the infection. process. Besides, it supports multiple hardware architectures, making the malware versatile to target various devices. However, the continuous evolution of these variants did not stop IoT devices from being one of the most security risks today, particularly within IoT-based smart grids. 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 preconfigured list of 62 credentials. Upon a successful login, the infected device's IP and credentials are reported to a hardcoded server.
A different loader program logs into the vulnerable device, identifies its system environment and downloads and executes the malware. Once infected, Mirai cloaks itself by deleting its binary file and renaming its process with a pseudo-random string. However, these infections do not persist over reboots. To make the infection more robust, Mirai kills other processes listening on TCP/22 or TCP/23 as well as those belonging to rival infections. After this, the bot listens for attack commands from its C2 server and resumes scanning for new victims.
The reason for this is that in order to spread further, Mirai searches for IoT devices that have open Telnet ports and attempts a bruteforce login into those devices. Once infected, the device will send its details back to the C2 server, instructing them to download the required malware binaries.
It is designed to run solely in memory; after execution, it deletes its binary file to avoid easy detection. It is possible for the botmaster to issue attack commands with target parameters. Mirai can run ten types of DDoS attacks. To date, more than 60 variants have been developed since the release of its source code and, once more, indicate the continuing security risks within the IoT ecosystem. 6, 7

# Chapter 3

# Countermeasures Against Mirai

## 3.1 Malware Analysis

Malware Analysis is a research area wherein malicious files are studied to understand various important aspects related to malware behavior, evolution, and target selection. This can enable security firms to enhance their strategies for defending against malware attacks. There are primarily three kinds of techniques for malware analysis: static, dynamic, and hybrid analysis.
Malware analysis may reveal how malware works and for questions such as: What is the operational procedure of malware? Which computers and applications are infected? What information is stolen or compromised? The two broad categories are static and dynamic analysis. Static analysis reviews malware without running the code, whereas dynamic analysis monitors behavior only when malware is running. Generally speaking, most analysts begin with simple static analysis techniques and move onto comprehensive dynamic analysis, which mostly involves reverse engineering and some specialized tools to interpret the malware formats. 8, 9

### 3.1.1 Static Analysis

Static analysis basically consists of the review of Portable Executable (PE) files without their execution. It helps in the identification of malware by detecting patterns such as API calls, string signatures, control flow graphs (CFG), opcode frequency, and byte sequence n-grams.

- Strings: Strings are fundamental for classifying the intent of the attacker because they carry most, if not all, of the semantic information.
- Control Flow Graph (CFG): It is a representation of the program structure that provides the control flow through nodes (code blocks) and edges (control paths).

CFG can portray malware behavior through representation based on the structure of the program.

- Opcodes: These represent the first part of machine code instructions executed by the CPU. The frequency of opcodes or similarity in their sequences is applied to malware detection.

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

Other static features include file size, function length, and networking aspects such as TCP/UDP ports and HTTP requests. 8

### 3.1.2 Dynamic Analysis

Dynamic analysis (also called behavior analysis) is based on the execution of suspicious files in controlled environments, such as virtual machines or emulators, observing their behavior. It is rather effective against all types of malware-both known and unknown, including obfuscated and polymorphic malware. On the other hand, when compared with static analysis, dynamic malware analysis requires more time and computational resources. Its common techniques involve function calls monitoring, parameters analysis, instructions tracking, and information flows following. API and system calls, as well as file system registry, and network features, are salient features involved in dynamic analysis.
Malware deploys anti-virtual machine and anti-emulator mechanisms in order to avoid detection by keeping their normal behavior once the presence of such environments is detected. In dynamic analysis, emulators, debuggers, simulators, and VMs are deployed, though some sophisticated malware can detect such controlled environments and may try to circumvent analysis. 8

### 3.1.3 Hybrid Analysis

Hybrid analysis combines static and dynamic methods, providing the advantages of both. While static analysis is fast, cheap, and safe, it can be easily evaded by obfuscation. Dynamic analysis, while more resource-intensive, is more reliable and hence capable of detecting variants and unknown malware. Hybrid analysis, by fusing both methods into one process, furthers malware detection accuracy. 8

## 3.2 Malware Detection

Malware detection methods can be categorized from different angles. In this section,two basic approaches are discussed: Signature-based and Heuristic-based detection.
In the early days, signature-based detection was used. This approach is efficient and very fast for known malware but cannot handle zero-day malware. Since then, some techniques such as behavior-based, heuristic-based, model-checking-based detection, etc have been proposed. Recently, some new approaches have been developed, namely deep

learning-based detection, cloud-based detection, mobile device-based detection, and IoT-based detection.
While behavioral and heuristic-based approaches can trace most types of malware, including new ones, they fail to detect all malware specimens. They are also challenging to devise a method which could trace a more complex malware specimen and find the unknown ones. 8, 9

### 3.2.1 Signature-based

Most antivirus software relies on the signature-based detection. Under this method of detection, the specific signatures or sequences of bytes, or file hashes get extracted from malicious files to detect similar malware. Signature-based detection has close to a zero percent rate of false positives but is susceptible to evasion by attackers that can easily modify malware's signature.
Signature-based detection is fast and effective for known malware but fails with new malware because of obfuscation techniques, such as 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.

However, due to its limitation of success to known malware, signature-based detection cannot provide sufficient solution against modern complex threats such as polymorphic malware. 8, 9

### 3.2.2 Behavioural-based

Behavior-based detection analyzes a file's runtime activities in order to identify malware. In the learning phase, patterns are extracted from the file, and during the testing phase, the file is classified as malicious or legitimate based on that pattern. It is capable of detecting unknown malware and also those malware that apply obfuscation techniques. However, it tends to have more false positives and can be computationally costly.
Heuristic-based methods mostly use data mining techniques such as Support Vector Machines, Naïve Bayes, Decision Trees, and Random Forests that help in malware detection by analyzing behaviour 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 good at detecting new families of malware, challenges such as handling large numbers of features, similarity detection, and inability of some malware to run in virtual environments remain. An increasing interest in machine learning and data mining techniques is playing an important role in improving malware detection by offering better understandability of features. 8, 9

# Part II

# Part Two

# Chapter 4

# Exploring Mirai: A Tool for Malware Analysis

## 4.1 Introduction

This Python script will sit at the intersection of binary analysis, system call tracing, and graphical representation of program flows. In fact, this script will use a variety of tools and libraries to achieve its tasks, namely r2pipe, networkx, pygraphviz, and matplotlib. It will look into an ARM32 binary for analyzing it and try to extract some system calls and show relationships between functions and strings within a binary in directed graph form.
This chapter explains the main components of the script, the libraries it makes use of, and how they work in harmony to extract meaningful insight from ARM binaries.

## 4.2 Overview of Script Objectives

The following are the main goals the script has:

1. **Binary Disassembly and Analysis**: Utilize r2pipe to deeply analyze an ARM binary and gather essential disassembled information from it.

2. **System Call Identification**: Parse ARM32 system calls and map them to respective function addresses.

3. **String Extraction and Mapping**: It finds and maps the strings inside the binary to functions to ease the process of reverse engineering and vulnerability analysis.

4. **Graph Construction and Network Analysis**: It uses networkx and pygraphviz in order to build and analyze a directed graph of function dependencies and their relationships, and then performs centrality and community detection studies.

## 4.3 Tools and Libraries Used

The script utilizes several third-party libraries, focusing on binary analysis and graph generation:

- **r2pipe**: Python binding for radare2 - a well-known framework for reverse engineering and analyzing binaries. It allows the script to interact with the binary - pulling relevant data such as what system calls are in use - and it automatically disassembles the binary.
- **pygraphviz**: Python interface to Graphviz graph layout and visualization software. Used to create and visualize the relationships that exist between functions and strings.
- **networkx**: Creation, manipulation, and study of complex networks. of nodes and edges. This is important in constructing the call graphs and the visualization of the binary flow.
- **matplotlib**: This is a standard Python plotting library used to visualize graphs coming from networkx.

These libraries will be the foundation of this script for automatic binary dissection, data structure creation, and visualization.

## 4.4 Binary Analysis using r2pipe

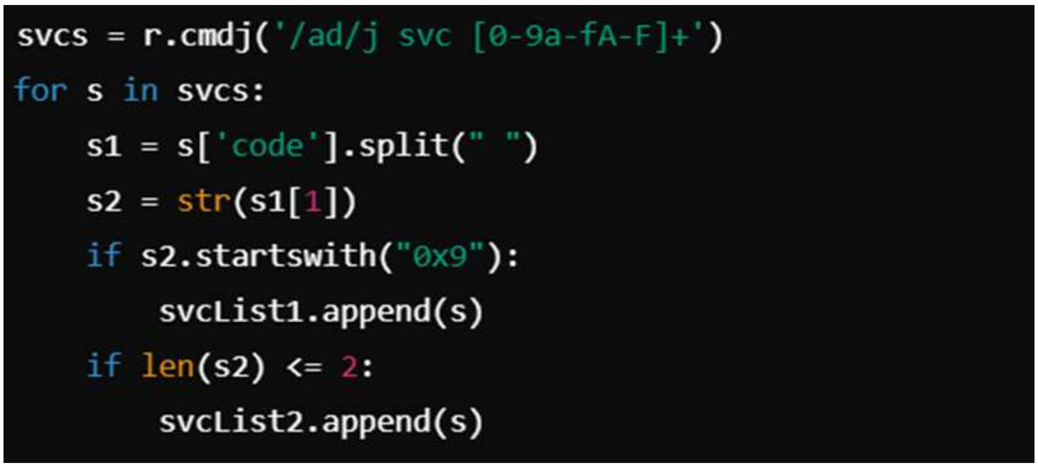
At the core of this script is the use of r2pipe, hooking up the script to the radare2. Radare2 is a powerful reverse engineering framework which is able to disassemble and analyze binary executables. The script opens a binary file specified via command-line argument using r2pipe.open(binary_path) and then runs the command aaa to analyze the binary, searching for functions, system calls, and any other interesting structures.

### 4.4.1 ARM32 System Call Table

The ARM32_SYSCALLS dictionary defines the mapping between the ARM32 system call number and system call name. This is later used to decode the system calls found inside the binary - from numeric values to human-readable names.
ARM32_SYSCALLS = 0: "restart_syscall", 1: "exit", ...
All system calls in the ARM's ISA are in this hash. As the script parses the binary, it refers to this hash for mapping numerical syscalls to their readable names.

### 4.4.2 System call extraction

The script then proceeds to parse the binary for system calls. Firstly, It uses regular expressions for look for special patterns within the disassembled code, notably supervisor calls - svc in the ARM architecture: (r.cmdj('/atj swi')) The equivalent in Radare2 for searching the 'svc' instructions. This script filters the interesting system calls into two lists according to certain criteria, including hexadecimal prefixes and code offsets.



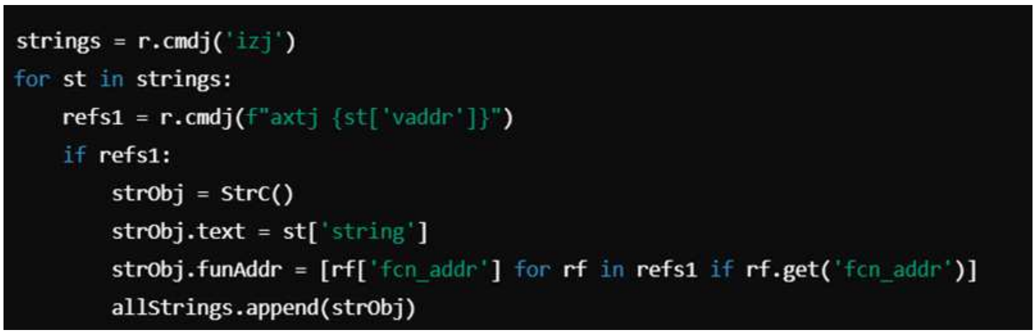
```python
svcs = r.cmdj('/ad/j svc [0-9a-fA-F]+')
for s in svcs:
    s1 = s['code'].split(" ")
    s2 = str(s1[1])
    if s2.startswith("0x9"):
        svcList1.append(s)
    if len(s2) <= 2:
        svcList2.append(s)
```

Figure 4.1. Snippet_1

From this data, it builds instances of a Syscall class: each holds details like hex value, offset in the binary, and the decoded system call name from the ARM32_SYSCALLS dictionary.

### 4.4.3 String Analysis

The script does some string extraction also with help of the izj command inside radare2. Those extracted strings from the binary are sourced into the instances of the **StrC** class ("StringClass"), along with the references of the functions that use those strings, (axtj command. This becomes a very important information later when its time to visualize what strings are associated with what functions.

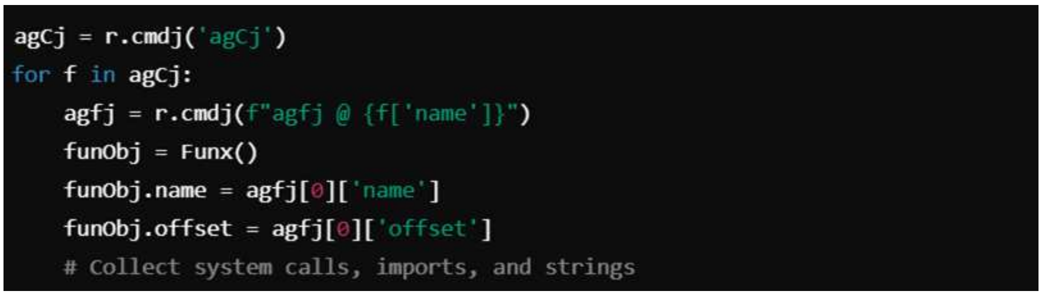


```python
strings = r.cmdj('izj')
for st in strings:
    refs1 = r.cmdj(f"axtj {st['vaddr']}")
    if refs1:
        strObj = StrC()
        strObj.text = st['string']
        strObj.funAddr = [rf['fcn_addr'] for rf in refs1 if rf.get('fcn_addr')]
        allStrings.append(strObj)
```

Figure 4.2. Snippet_2

### 4.4.4 Function Analysis and Call Graph Construction

This script will start by gathering information about the functions in the binary. It achieves this by running radare2's graph analysis commands: (agCj, agfj). For every function that is found, it finds any system calls it makes, strings it uses, and its imports. The information all gets stored in objects 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
```

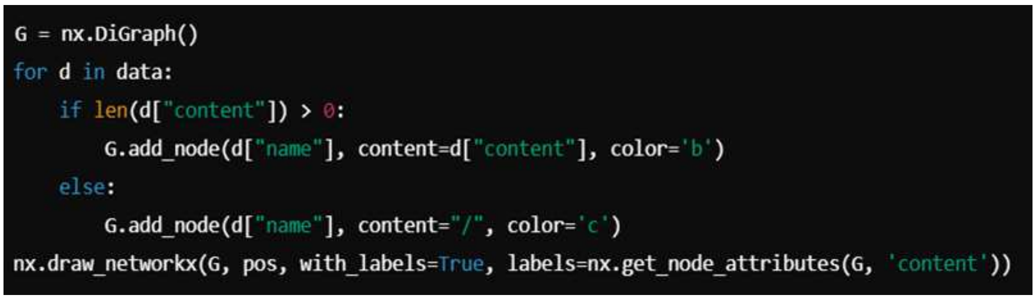
Figure 4.3. Snippet_2

## 4.5 Graph Visualization

Now that functions, system calls, and strings have been extracted, the script uses **NetworkX** to build a directed graph (nx.DiGraph()), showing functions or strings as nodes, and the edges will represent their relationships.
The color of the nodes depends on their type:

- **Functions**: Blue or Cyan depending on whether there are system calls or not
- **Strings**: Red

Spring_layout is used to compute the position of the nodes. **Matplotlib** is used to actually draw the graph, with labels indicating the content of the nodes (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

This script saves the call graph along information in 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 usign the NetworkX functions are the following:

- **Centrality Measures**: Betweenness, closeness and degree centralities are computed to identify key functions within the graph. These metrics will provide the importance and influence of functions in the binary.

- **PageRank Analysis**: This will calculate the PageRank scores to rank functions by their interconnectivity, thus providing an overview of critical components.

- **Community Detection**: Using the Louvain method for community detection, this script will group related functions together; it also shows modular or cohesive functionality.

For all these, we will create a dedicated subgraph from the full graph by removing only the string nodes.

### 4.6.1 Closeness centrality

**Closeness centrality** refers to how close a node is with other nodes in the network. This concept is based on the notion of the average shortest path, the so-called "geodesic path," from any given node to all others. First, the mean distance to other nodes is calculated for each node. The nodes which are on average closer to others have lower values, and these nodes can typically access information more easily and have a greater influence on other nodes.

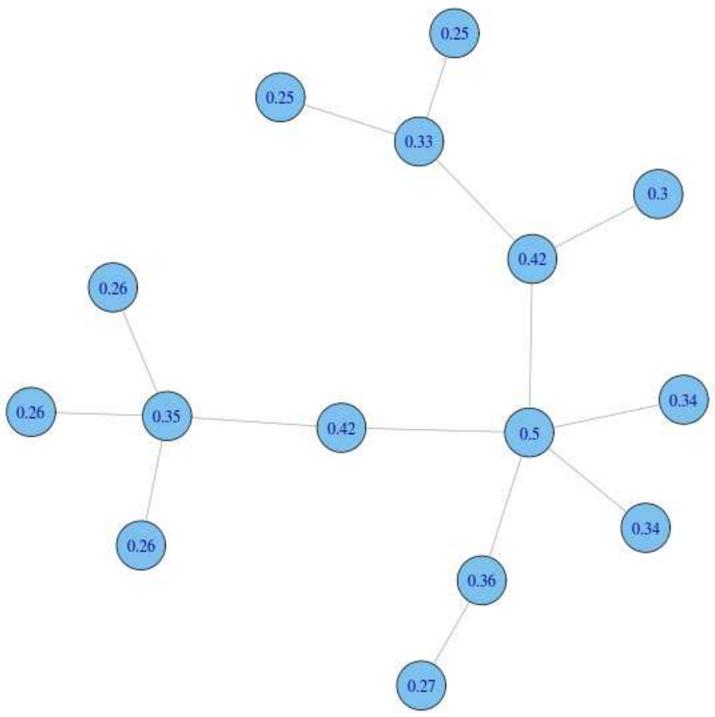


Figure 4.5. Example of closeness centrality

For intuitive understanding of that measure, its inverse is usually used, giving higher values to more central nodes. It is called closeness centrality and calculated as the inverse of the average shortest path distance for a node. Essentially, nodes with higher closeness centrality can reach other nodes more quickly.
Unlike the degree or eigenvector centrality, closeness centrality is a measure of how central an actor is with respect to the geodesic distance, rather than the number of direct links. 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 shares some weaknesses: for instance, an unreachable node-in a disjoint graph-has infinite distance, and therefore a closeness centrality of zero. Such nodes are usually excluded from centrality calculation, focusing on the biggest connected component, or assign a large distance for unreachable pairs in order to make calculations feasible.
First, there is the problem that closeness centrality values are usually bunched together

within an extremely small range, which can make it difficult to clearly distinguish highly central nodes from their less central counterparts. Even small changes in network structure can result in dramatic changes in closeness ranking of nodes. 10
The closeness centrality function in NetworkX can be used for malware function analysis in a system call or behavioral graph. Centrality defines a measure of how "close" a node-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. In malware analysis, it will highlight the functions that are central to the general course of execution of the malware.
Function examples with high closeness centrality are excellent targets for security, as their failure could disrupt the whole malware’s operations.
Using the closeness_centrality() function provided by NetworkX will enable us to find which malware functions are important in the overall execution of malware. More importantly, it could be useful for prioritizing remediation strategies in disrupting the most important parts of the malware behavior. Targeting high closeness centrality functions may potentially disrupt the very core operation of the malware in question and limit its propagation or effects.
In our case study, after executing the script on a large set of Mirai samples, the function results indicate that the top-ranking system calls are: **kill**, **close**, **write**, **futex**. These syscalls hold the highest closeness centrality values.
The reason behind this could be that, since Mirai performs many network operations (opening and closing sockets), "close" and "open" are frequently invoked after/before a sequence of network-related syscalls. "futex", instead, might acts as a bridge in the malware’s control flow when synchronizing threads. Its usage pattern in multi-threaded environments places it at a point in the syscall graph where it is equidistant to many other operations, hence a high closeness centrality.

### 4.6.2 Betweenness centrality

**Betweenness centrality** quantifies the number of times a node lies on the shortest paths between other pairs of nodes in a network. A high betweenness central node has a great deal of control since it sits on many of the paths between other pairs of nodes. It also means that such nodes are important in terms of communication and the removal of such nodes from a network would badly hinder the network.

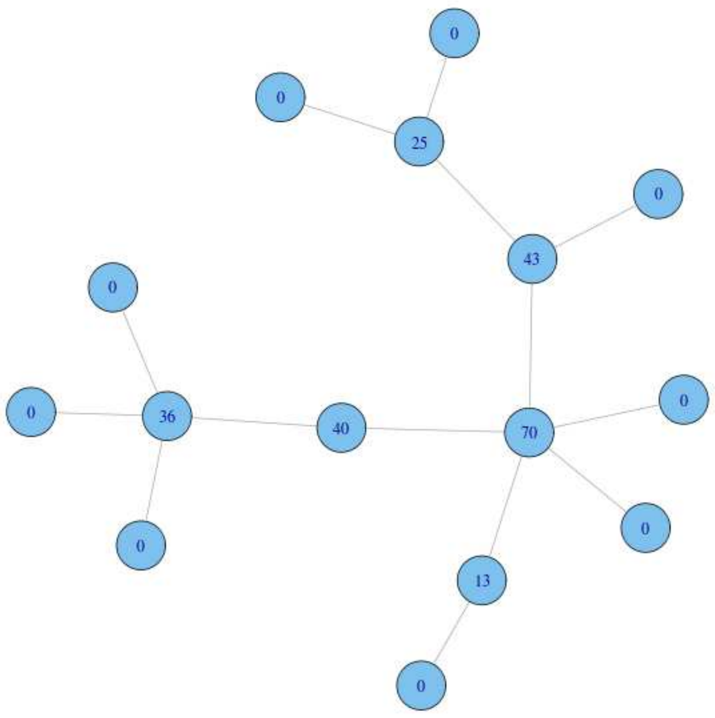


Figure 4.6. Example of betweenness centrality

Centrality is mathematically defined as the number of shortest paths between pairs of nodes passing through a node, divided by the total of the number of those pairs of shortest paths. If no shortest paths pass through a node, then that node has zero betweenness contribution. This centrality differs from other centralities, such as degree or closeness centrality. A node may have a low number of direct connections. It doesn't need to have a high degree nor be close to others, yet still it may achieve a high degree of betweenness if it is in a position of bridging between groups of nodes. Such vertices are sometimes called "brokers".
That means this would correspond to the case of a star network where the highest betweenness is for the central node joining all the others, while the minimum betweenness is for so-called leaf nodes, connected with the network by just one edge.
Unlike closeness centrality, betweenness centrality values usually span over a wide range, and thus, it might be easier to tell which nodes are most influential in a network. In practice, betweenness centrality calculation involves a computation of shortest paths between

any pairs of nodes, which may be computationally intensive but can handle in a sparse network where the total number of links is rather low compared to the total number of nodes. [11]

The betweenness centrality function from NetworkX can be useful to analyze malware functions in either a system call or behavior graph, which indicates key points of control or influence within the flow of execution of malware. Betweenness centrality measures how frequently a node-malware function shows up along the shortest paths available between other pairs of nodes. This will help during malware analysis in highlighting functions that act as bottlenecks or crucial intermediaries in malware behavior.

Generally speaking, functions with high betweenness centrality usually represent important "intermediaries" between the different phases of the malware operation..

The identification of a bottleneck within the malware execution path may indicate places that are important for the rupture of malware functionality. It is foreseen that neutralizing or isolating those functions with high betweenness centrality might break the core operations of malware.

Betweenness centrality will helps, in malware analysis, highlighting important "chokepoints" in malware flow, which are those functions that are major conduits and will heavily impact malware's capability to perform or propagate. The ability of an analyst to disrupt such high-betweenness functionality will highly hinder the malware's ability to operate and is a good tool for prioritizing mitigation efforts.

In our case study, after executing the script on a large set of Mirai samples, the function results indicate that the top-ranking system calls are: **fcntl**, **open**, **close** and **futex**. These syscalls hold the highest betweenness centrality values.

An explanation could be that "fcntl" can be used to manipulate file descriptors, such as setting non-blocking mode on sockets, making it a critical bridging step between resource acquisition and data operations. This intermediary role gives fcntl high betweenness centrality. While for "open", because it acts as a gateway for accessing resources, it frequently lies on the paths of different execution flows, especially when setting up for attacks or logging. This makes it another high betweenness node.

### 4.6.3 Degree centrality

**Degree centrality** for a node is defined as the fraction of nodes it is connected to. [12]

For both directed and undirected networks, the degree of centrality refers to the connectedness of nodes through several edges. In the directed network, the node's Centrality entails the summation of the total number of incoming and outgoing links. In the case of undirected networks, it entails the summation of the links. Centrality degree is the measure of influence of the node directly in the local network.

This is because the nodes that have the most incredible power of centrality have a greater influence directly on the nodes they are connected to. Centrality can also be used to determine those influential nodes that lie in the shortest distance between two nodes that connect two different components of a network. [13]

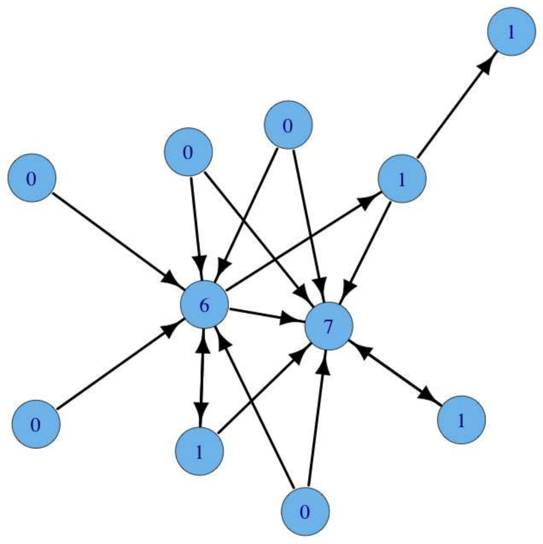


Figure 4.7. Example of degree centrality

The degree centrality function from NetworkX is used to compute the centrality of nodes based on the number of their connections or edges in a graph. It acts like a strong analytical tool that would most likely point towards the most influential or critical functions within the flow of malware execution and highlight which of them are used most.
Centrality in a network could reveal the structure of malware communication-for example, high centrality functions could express the frequency in which certain functions are communicating with each other.
This function is useful within malware function analysis; it identifies the most central and connected functions within the malware execution graph. In general, highly central functions will be crucial to the malware's operations and thus represent key points for both reverse engineering and mitigation.
In our case study, after executing the script on a large set of Mirai samples, the function results indicate that the top-ranking system calls are: **close**, **rt_sigprocmask**, **rt_sigaction** and **futex**.
This could be attributed to the fact that "close" is being frequently used in many stages of the malware's life cycle, such as scanning, communication, and attack execution, thus involving a great deal of interaction and, hence, high centrality. The high centrality of the "Futex" syscall is explained by it being an integral part of the thread lifecycle and, hence, very frequently called in parallel with other syscalls, like "nanosleep", in the context of handling threads.
"rt_sigaction" is usually used in conjunction with other syscalls that may raise signals, like

"kill", "alarm", or "setitimer". Also, when the malware is doing operations that should not be interrupted, it may block signals using "rt_sigprocmask". This syscall usually connects to other threading and I/O-related syscalls, hence its high degree centrality.

### 4.6.4 Pagerank

**PageRank** is an algorithm developed at Google to make decisions on the importance of a web page. Actually, the naming was done after one of Google's cofounders, Larry Page. If a page has many links to it from other important Web pages, then that page is likely to be important too. In other words, this algorithm estimates the importance of a page by counting its incoming links and their quality.

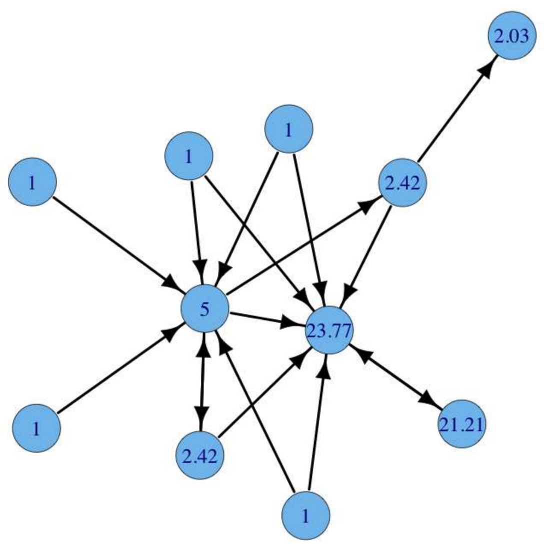


Figure 4.8. Example of pagerank

In simplified form, PageRank would function by simulating a random user who clicks through the links on the web. With every page the user gets to, the importance of that page is increased depending on all the pages that linked to that page, considering their PageRank values. The more quality links point to the page, the more PageRank will be earned.
PageRank initiates with the same score for all pages but then iteratively refines this ranking. At each step, a page spreads its PageRank over all pages it links to. If a page has two outgoing links, it will distribute the score across those two. Pages that acquire links from lots of other important pages tend to increase in PageRank over time, whereas pages

with fewer links, or with less important links, decrease in score.
The PageRank formula involves a damping factor, representing the probability that eventually a user will continue randomly clicking and will end up on some other page. This, to prevent this system giving it too much value, and to ensure convergence in this algorithm. 14
The NetworkX page_rank() function can be a very able tool when analyzing malware functions in the graph of system calls. It can be used in ranking malware functions in order of their importance in the flow of executions.
In malware such as botnets or ransomware, there are a few functions that everything seems to center on. PageRank can help in identifying these critical functions serving at important points in the malware's operation.
This would give an in-depth understanding of the critical functions or system calls inside the malware, hence helping the analyst understand its structure and prioritize the efforts for mitigation. By highlighting the most influential points, PageRank enables the analysis to focus on malware analysis and remediation.
In our case study, after executing the script on a large set of Mirai samples, the function results indicate that the top-ranking system calls are: **kill**, **futex** and **write**.
The **kill** syscall is used by Mirai primarily for process management purpose. It attempts to terminate other malicious processes running on the same device to gain exclusive control. It also checks for and terminate any previous instances of itself to prevent duplication and avoid conflicts. Since Mirai uses this syscall frequently to eliminate competition and manage its own instances, it gets a high PageRank. **Futex** is used for thread synchronization in multi-threaded applications. Mirai uses futex to handle synchronization between threads, particularly when launching a DDoS attack, which often involves creating multiple threads for sending packets simultaneously. The malware might use multi-threading to handle different tasks like scanning for vulnerable devices, connecting to command-and-control (C2) servers, and executing attack commands concurrently. The **write** syscall is used for output operations, particularly to send attack payloads during DDoS operations. This can include sending UDP, TCP, or HTTP flood packets.

### 4.6.5 Louvain communities

The Louvain method is a technique that tries to detect communities in large networks by maximizing a modularity score. The latter is a measure of the quality of the grouping of the nodes into communities with respect to a random arrangement. It basically searches for clusters of nodes which are more internally connected than with the rest of the network. The Louvain algorithm operates hierarchically and consists of two main iterative phases:

1. **Local Movement of Nodes**: Each node in the network is initially assigned to its own community. The algorithm then considers for each node whether its removal from its own community to an adjacent one would result in increased modularity. If the removal of a node to the other community produces a positive gain in modularity, it is reassigned to that community. This procedure goes forward for all nodes in successive iterations until no more relocations may improve the modularity, reaching a local maximum.

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 iteratively repeated until no more improvement in modularity is possible. This yields a hierarchical structure of the communities as well. The Louvain method is very popular since it is rather fast and easy to realize, but on the other side, memory-consuming while storing network data. 15, 16, 17

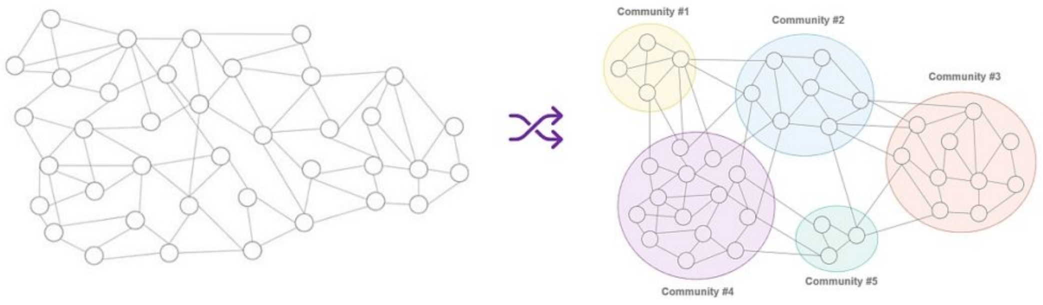


Figure 4.9. Example of Louvain communities algorithm

The community detection in NetworkX is done via the function louvain_communities(), which applies the Louvain method to perform its analysis. In malware system call analysis, the function enables the discovery of clusters of highly connected system calls, which may represent different functional modules or behaviors of malware. This function is used to find out clusters or communities in the syscall graph. Each community that can be found represents groups of highly frequent co-occurring syscalls or those that are functionally related to each other. 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, mechanisms of persistence, or evasion techniques.
- **Detection and Mitigation**: Focusing on critical syscall groups for designing more specific detection protocols or mitigating strategies.

- **Prioritization for Further Analysis**: More central or densely connected syscall communities may be more critical to the malware's operation. Analysts can focus on these areas for further 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 is a comprehensive toolkit for malware analysis in the ARM32 architecture. This script automates the extraction of system calls, functions, and strings, mapping their interactions in a graph that enables the cybersecurity professional to efficiently analyze malware. The visualization of the binary's behavior and the use of NetworkX's most useful functions give deep insights into the operation of the malware; this script is a very powerful tool in identifying, understanding, and mitigating malicious threats.
This script has enormous potential for various future developments that can enhance its capabilities of malware analysis. Given the dynamic nature of the cyberattacks landscape and constantly improving malware, there are various areas where this script can be further enhanced to provide more profound insights and more successful analysis of malware binaries. The following section will explore some possible future developments by enhancing automation and integrating with state-of-the-art techniques for the expansion of the scope of analysis.

- **Machine Learning Integration for Automated Malware Classification**: One potential enhancement is using machine learning techniques for classification, based on the patterns that were identified in the binary malware analysis. It can be trained on large sets of data featuring known malware variants to enable self-classification of new variants into their respective known families, or to make out new patterns that might indicate previously unseen threats. Machine learning models would be trained to recognize malicious patterns based on system calls, control flow graphs, and function behavior.

- **Expanding Platform Support Beyond ARM32**: Whereas the script at the moment is cut for ARM32 binaries, support for more architectures and operating systems would greatly extend the places where this script could be applied. As different variants of malware have rised up, targeting other platforms like x86, x64, MIPS, and RISC-V, increasing cross-platform compatibility with regard to the script's analysis of malware would greatly increase its versatility for a wider number of environments. It would also make the script more applicable to a wider circle of analysts, who could thereby identify malware that attacks systems other than ARM32-based IoT and embedded devices.

- **Improved Visualization and Reporting**: The graph-based visualization of function relationships and system calls is already a very strong tool for malware analysis, but it needs further extension with more detailed and customizable visualizations, as well as automated reporting features. Richer insights would result from advanced graph analytics, thus improving the experience of the users: would make the script more accessible to analysts of all skill levels. Improved visualization would serve to make interpretations of complex malware behaviors more readily attainable, and automated reporting would ease the process of documenting results for incident response or forensic investigations.

- **Enhancement of Dynamic Analysis Capabilities**: Currently, the script mainly deals with static analysis of binaries, examining the code without running it. Adding dynamic analysis—run malware in a controlled environment or sandbox—would give the capability for the script to observe real-time behavior of the malware and sometimes give insights that may not be caught through static analysis alone. Dynamic testing would offer a view complementary to that of malware and enable the script to pick up behaviors that only get triggered in certain conditions or during runtime.

The Python script has ample scope for future enhancements that would make it an even better helper tool for malware analysis. Firstly, by incorporation of machine learning into its core for automated classification, then enhancing dynamic analysis, followed by support for more platforms, and lastly by improvement in the recreation of data. Some areas where this tool can further evolve are in visualizations and collaboration with threat intelligence platforms. As malware becomes more sophisticated, sophisticated, and diversified across platforms, extending the script's capabilities through such proposals will aid security analysts to move ahead of emerging threats and make it a valuable tool worth having in the cybersecurity defense for many years.

# 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/louvain/.

17. URL https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.community.louvain.louvain_communities.html#networkx.algorithms.community.louvain.louvain_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.

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.