Masterarbeit

Predicting SSH keys in Open SSH Memory dumps

A report by

Rascoussier, Florian Guillaume Pierre

Matrikelnummer (Passau): 112485
Matrikelnummer (INSA): 4018543

Erstprüfer
Prof. Dr. Michael Granitzer
Zweitprüfer
Prof. Dr. Harald Kosch

Betreuer
Christofer Fellicious
Prof. Dr. Pierre-Edouard Portier
Prof. Dr. Elöd Egyed-Zsigmond

October 26, 2023

Abstract

As the digital landscape evolves, cybersecurity has become an indispensable focus of IT systems. Its ever-escalating challenges have amplified the importance of digital forensics, particularly in the analysis of heap dumps from main memory. In this context, the Secure Shell protocol (SSH) designed for encrypted communications, serves as both a safeguard and a potential veil for malicious activities. This research project focuses on predicting SSH keys in OpenSSH memory dumps, aiming to enhance protective measures against illicit access and enable the development of advanced security frameworks or tools like honeypots.

This Masterarbeit is situated within the broader SmartVMI project, a collaborative research initiative with the objective to advance artificial intelligence-based mechanisms for attack detection and digital forensics. Specifically, this work seeks to build upon existing research on key prediction in OpenSSH heap dumps. Utilizing machine learning algorithms, the study aims to refine feature extraction techniques and explore innovative methods for effective key detection prediction. The objective is to accurately predict the presence and location of SSH keys within memory dumps. This work builds upon, and aims to enhance, the foundations laid by SSHkex [1] and SmartKex [6], enriching both the methodology and the results of the original research while exploring the untapped potential of newly proposed approaches.

This report encapsulates the progress of a year-long Master's thesis research project executed between October 2022 and October 2023. Conducted within the framework of the PhDTrack program between the University of Passau and INSA Lyon, the research has been supervised by Christofer Fellicious and Prof. Dr. Michael Granitzer from the University of Passau, as well as Prof. Dr. Pierre-Edouard Portier from INSA Lyon. It offers an in-depth discussion on the current state-of-the-art in key prediction for OpenSSH memory dumps, research questions, experimental setups, programs development as well as discussing potential future directions.

Acknowledgements

A special acknowledgment goes to Christofer Fellicious, my engaged supervisor at the University of Passau, for his guidance, support and feedback during the Masterarbeit.

I want to express my sincere gratitude to my colleague and friend, Clément Lahoche, whose human and technical skills have been a great source of inspiration and motivation throughout this project; especially considering that we have been working on closely related subjects. It has been a great pleasure to share our ideas and insights, and to collaborate on the development of several programs necessary for the experimentations.

Another acknowledgments go to my esteemed supervisors Prof. Dr. Granitzer and Prof. Dr. Portier for their support and feedback during the Masterarbeit.

I would also like to express my sincere gratitude to all the persons that have helped me, even punctually, during the Masterarbeit with their valuable help, insights, discussions and contributions as well as all the persons involved in the PhDTrack program that made this Masterarbeit possible, including but not limited to:

- Lionel Brunie, Director of CS Department at INSA Lyon, that makes this PhDTrack program possible from the French side.
- Harald Kosch, Head of the Chair of Distributed Information Systems at the University of Passau, that makes this PhDTrack program possible from the German side.
- Natalia Lucari, PhDTrack coordinator at INSA Lyon, for her support and help during the PhDTrack program.
- Ophelie Coueffe, PhDTrack coordinator at the University of Passau, for her support and help during the PhDTrack program.
- Elöd Egyed-Zsigmond, PhDTrack coordinator at the University of Passau, for the subject selection and administrative support.
- All the other PhDTrack students for the great atmosphere, mutual help and the interesting discussions during almost two years.

Finally, my last acknowledgments go to my family and friends for their support and encouragements.

Contents

1 Introduction

The digital age has brought with it an unprecedented increase in the volume and complexity of data that is being generated, stored, and processed. This data is often sensitive in nature, and its security is of paramount importance, making cybersecurity a critical focus area. This evolving landscape is fraught with challenges that continue to amplify the importance of digital forensics in IT systems. One area that stands out for its widespread use and importance is the Secure Shell protocol (SSH) and its most popular implementation, OpenSSH. SSH is a cryptographic network protocol widely used for secure remote access to systems. It is also used for secure file transfer, and as a secure tunnel for other applications. SSH is a key component of IT systems whose encryption capabilities are critical to the security of IT systems. However, it also presents a unique set of challenges, most notably the concealment of malicious activities.

A common case is when a unauthorized actor gains access to SSH keys so as to get access to a system. This can happen through a malicious human actor, but more commonly through automated processes such as malwares and botnets. This situations present a formidable and growing threat to cybersecurity, affecting a broad range of stakeholders from governments and financial institutions to individual users. In just 2019, the number of Command and Control (C&C) servers for botnets increased by 71.5%, leading to an estimated $19 billion in advertising theft [11]. Many malwares and botnets «have in common that they have used as attack vector the Secure Shell (SSH) remote access service» [11].

At the heart of the issue lies the fact that SSH veils its communications through encryption, making it difficult to detect malicious activities. To be able to detect those potential malicious actors, it is possible to replace SSH by a honeypot that enable to monitor pseudo-SSH activities. There is a range of readily available honeypots, such as Kippo or Cowrie, which are designed to emulate a vulnerable SSH system and attract attackers [12]. The problem lies that thoses honeypots are not able to mimic perfectly a real system, which makes them easy to detect by experienced attackers. As stated by „Analysis of SSH Honeypot Effectiveness“: «The ability to collect meaningful malware from attackers depends on how the attackers receive the honeypot. Most attackers fingerprint targets before they launch their attack, so it would be very beneficial for security researchers to understand how to hide honeypots from fingerprinting and trick the attackers into depositing malware. [...] What is certain is that if a cautious attacker believes they are in a honeypot, they will leave without depositing malware onto the system, which reduces the effectiveness of the honeypot» [13].

The are other approaches that allows to decrypt SSH connections without relying on a honeypot, like the *man-in-the-middle* or *binary manipulation* with their own set of challenges [1]. Instead of relying on softwares that mimics or modify a real system, it is possible to use a real unmodified system directly. The idea is to be able to decrypt SSH connection channels, which is possible if the SSH keys are known. Since SSH encryption keys are typically stored in the main memory of a system, it is possible for the administrators to extract them through the exploitation of memory dumps of a targeted system. In this context, the ability to detect SSH keys in memory dumps, and specifically OpenSSH keys, is critical to the development of effective SSH honeypot-like systems. The research introduced by the SmartVMI project with SSHKex, SmartKex, the present thesis and the future related work could be used to develop such a new type of system-monitoring tools. This new kind of tools would be very difficult to detect by attackers, increasing their effectiveness, and wouldn’t require the alteration of the system. The present report is focused on the SSH key detection in memory dumps, which is a key component allowing to decode SSH communications such that it become possible to intercept malicious communications and to detect malicious

activities.

1.1 Research Questions

At the very beginning of this thesis, first questions were:

- What is the state of the art in the field of security key detection in heap dump memory?
- What are the challenges of security key detection in heap dump memory?
- How can the existing methods for detecting SSH keys in OpenSSH heap dumps be improved?

The SmartVMI project has already made significant progress in the detection of SSH keys in OpenSSH heap dumps. An open dataset of memory dumps has been created, and a simple yet effective method for detecting SSH keys has been developed. The dataset has been used to train and test simple machine learning algorithms, and the results have been promising. The research has been published in the form of two papers, SSHkex [1] and SmartKex [6], which is the basis of this thesis.

However, there is still room for improvement, particularly in the area of machine learning algorithms. This thesis seeks to build upon the existing research by refining feature extraction techniques and exploring innovative methods for effective key detection prediction. The objective is to accurately predict the presence and location of SSH keys within memory dumps. Rooted in this context, this Masterarbeit aims to address several key research questions:

- **Memory graph:** How can we develop a memory graph representation to improve the prediction of SSH keys in memory dumps?
- **Memory graph embedding:** How can we develop a memory graph embedding representation to improve the prediction of SSH keys in memory dumps?
- **Feature importance:** What features are most indicative of SSH keys in memory dumps?
- **Feature extraction:** How can these features be extracted from memory dumps and used to train machine learning algorithms?
- **ML for key prediction:** How can machine learning algorithms be optimized for the prediction of SSH keys in memory dumps?
- **Graph Convolutional Networks for key prediction:** How can GCN be used to improve the prediction of SSH keys in memory dumps, and how does it compare to traditional machine learning algorithms?

By tackling these research questions, this thesis seeks not only to advance the academic understanding of SSH key prediction and digital forensics but also to provide practical insights that could lead to the development of more secure and effective systems.

1.2 Commitment to Open Science and Reproducibility

In alignment with the principles of Open Science, this thesis aims to be not just a scholarly report but also a comprehensive guide for anyone who wishes to understand, replicate, or extend the work presented. Open Science is a movement that advocates for the transparent and accessible sharing of scientific research, data, and dissemination processes [9]. It is built on six fundamental principles [8]:

1. **Open Methodology**: Detailed methodologies are provided to ensure that the experiments can be replicated.
2. **Open Source**: All code used in this research is available for scrutiny and reuse. As such, all code including the LaTeX code for the present report [1] is properly documented and can be accessed on GitHub.
3. **Open Data**: Raw data and the data processing techniques are made publicly available.
4. **Open Access**: The research is published in a manner that is free for all to read and download.
5. **Open Peer Review**: The peer review process is transparent. In the case of this Masterarbeit, the research is reviewed by the supervisors of the project.
6. **Open Educational Resources**: Any educational content produced is shared openly.

To ensure the highest level of reproducibility and accessibility, this thesis includes what might seem like exhaustive details, such as hardware or software specifications, precise shell commands and some code implementations used during the research. These are included to provide a complete picture and to minimize the friction for those who wish to replicate the experiments, whatever their level of expertise may be. By adhering to the principles of Open Science, this thesis aims to contribute to a more transparent, collaborative, and efficient scientific community.

1.2.1 GitHub Repositories

In the context of the present Masterarbeit, a number of GitHub repositories have been created to facilitate the sharing of code and data. These repositories are listed below:

- **masterarbeit_report_onyr**: Repository containing the LaTeX code for the report as well as several scripts related to dataset exploration: `https://github.com/passau-masterarbeit-2023/masterarbeit_report_onyr`
- **mem2graph**: Memory graph creation utility built in Rust, featuring different graph creation and embedding strategies. Collaboration with Clément Lahoche: `https://github.com/passau-masterarbeit-2023/mem2graph`
- **research-base**: Custom Python framework for developing programs that include all the basics of a research project, such as logging, environment and argument loading, result keeping, and more. Collaboration with Clément Lahoche: `https://github.com/0nyr/research-base`

[1] The present report repository can be found here: `https://github.com/0nyr/masterarbeit\_report`

- **data_processing**: Python program for data processing and machine learning for SSH key prediction. This repository contains tests on machine learning model training and evaluation for classical .csv based embedding files from *mem2graph*: `https://github.com/passau-masterarbeit-2023/data_processing`
- **phdtrack_project_3**: Legacy repository containing the first version of the memory graph creation utility and the first version of the dataset creation script. Collaboration with Clément Lahoche. `https://github.com/0nyr/phdtrack_project_3`
- **memory_graph_gcn**: Main Python program and scripts around GCN for SSH key prediction. This program leverages the modified DOT file with embedding generayted by *mem2graph*: *mem2graph*: `https://github.com/passau-masterarbeit-2023/memory_graph_gcn`
- **phdtrack_server_scripts**: Scripts for the servers used for computating. This repository contains the scripts used to install the necessary tooling and run the experiments on the different servers we used. Collaboration with Clément Lahoche: `https://github.com/passau-masterarbeit-2023/phdtrack_server_scripts`

As one can see, and considering the collaborative work effort that has been, it has been decided to regroup all repositories related to the OpenSSH heap dump exploration in a single GitHub organization, *passau-masterarbeit-2023* `https://github.com/passau-masterarbeit-2023`.

1.3 Structure of the Thesis

2 Methods

In the preceding chapters, all the necessary background knowledge to understand the methods have been introduced. In this chapter, we will present an overview of the challenges, the methods, and the tools we have developed for this thesis. We will first describe the dataset we have used. Then, we will describe the programs developed for this thesis. Finally, we will describe how we have packaged and deployed our programs with Nix.

2.1 Hardware and software architecture

Throughtout this thesis, we have used a variety of hardware and software architectures.

2.1.1 Hardware development and testing environment

In this section, as a reference for the reader, we will describe shortly the hardware development environment. All environments are running some Linux `x86_64` distribution.

At the start of the project, around the end of 2022, the project started on a old laptop *HP EliteBook Folio 1040 G2*, running `Ubuntu 22.04 LTS (Jammy Jellyfish)` with the following specifications:

- **CPU:** 5th Generation Intel Core i7-5600U 2.6 GHz (max turbo frequency 3.2-GHz), 4 MB L3 Cache, 15W
- **GPU:** Intel HD Graphics 5500
- **RAM:** 8GB DDR3L SDRAM (1600 MHz, 1.3v)

This device was used for the first experiments, and for the development of the first programs. However, it was not powerful enough to run the experiments on the whole dataset, and especially working on ML part. As such, we have moved to a more powerful machine, a *TUXEDO InfinityBook Pro 16 - Gen7* with the following specifications:

- **CPU:** 12th Gen Intel i7-12700H (20) @ 4.600GHz
- **GPU:** NVIDIA Geforce RTX 3070 Ti Laptop GPU
- **RAM:** 64GB DDR5 4800MHz Samsung

For the Operating System, we have switched from `Fedora 37` to `NixOS 23 (Tapir)`. This change was motivated by the fact that `NixOS` is a Linux distribution that uses a purely functional package management system [10]. This means that the operating system is built by the Nix package manager, using a declarative configuration language. This allows to have a reproducible development environment, and to easily switch between different development environments. This has proved to be very useful in many areas like work environment isolation, on work collaboration with Clément Lahoche, and for software deployment to the server.

Unfortunatly, the *TUXEDO InfinityBook Pro 16 - Gen7* laptop was not powerful enough to run the experiments on the whole dataset. Running the python script would have taken more than a week for some simple ML experiments to run on the whole dataset, and even reprogrammed and better optimized programs in Rust that we have tested would take more that 10 hours to process the dataset. Small bash and python scripts have been run on this laptop, but all the main experiments have been run on the server.

In that context, we were provided a development server towards the end of the thesis, around august 2023. This server is a *AS-4124GS-TNR* with the following specifications:

- **CPU:** 2x AMD EPYC 7662 (256) @ 2.000GHz
- **GPU:** NVIDIA Geforce RTX 3090 Ti
- **RAM:** 512GB DDR4 3200MHz

This server is running `Ubuntu 20.04.6 LTS` and is used to run the ML experiments on the dataset, as it is much more powerful than the laptop. This server has be provided by the Department of Computer Science of Universität Passau, and especially the Chair of Data Science of Prof. Dr. Michael Granitzer. We would like to thank them for their support.

2.1.2 Software, languages and tools

In Computer Sciences, it does't take long to realize that testing hypotheses, diving deeper in problems and finding solutions to them is a very iterative process that requires a lot of experimentation. As such, the development of scripts and programs has been a substantial part of this thesis, from the very beginning to the very end. In this process, we have used a variety of tools and programming languages, such as Rust, Python, Bash, or Nix just to name the programming language used.

In this section, as a reference for the reader, we will describe the software architectures, languages and tools that have been used throughout this thesis.

Throughtout the project, we have come to use a range of programming languages. Initial tests have been done using shell and bash command and simple scripts. However, as the project grew, we quickly moved to more powerful programming languages.

Python version 3.11 has been the main language for high level data science and ML development. This new version of python features many improvements over the previous version, and especially in terms of performance. Better error messages, exception groups, improved support for f-strings, support for TOML configuration files, variadic generics, improved support for asyncio, and new modules for working with cryptography and machine learning are just some of the new features of this new version of python. While relatively new, this is why we have decided to use this version of python for the development of the ML part of the project.

Although Python is a popular and powerful language, it is not the most efficient language. As such, we have used Rust for some parts of the project, especially when no high level library is needed and when performances are critical to be able to parse efficiently the dataset. Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety. It is a very powerful language, and is especially useful for low-level programming. We have used it for the development of the algorithms that are used to extract the data from the dataset.

2.1.2.1 Packaging and deployment

We made an extensive use of git repositories for version control, with GitHub as a main platform for hosting the repositories. An ever growing number of script and programs have been developed for this thesis. As such, we have needed a way to easily deploy those programs on different machines.

Rust comes with a handful of tools for managing packages and dependencies. Cargo is Rust's build system and package manager. Cargo downloads your Rust project's dependencies, compiles your project, makes executables, and runs the tests. It is a powerful tool that allows to easily manage Rust projects. However, it is not the best tool for deploying programs on different machines.

On Python's side of things, things are a bit more complicated. For a long time, we have relied on virtual environments using the `conda` package manager. However, it is heavy to use, and it doesn't allow to easily export an environment from one linux distribution to the other. This was especially a problem when development was done on a Fedora system but the server was running Ubuntu.

An example is the library `pygraphviz`. This library relies on third parties system libraries, that have different names depending on the linux distribution:

- **Ubuntu:** `sudo apt-get install graphviz graphviz-dev` is needed before a correct install of the Python `pygraphviz` library.
- **Fedora:** `sudo dnf install graphviz graphviz-devel` is needed before installing `pygraphviz`

Although it can be seen as a minor issue, this is just one example among dozens of libraries. This is real problem with `conda`. This is why we have decided to use Nix for managing python packages and dependencies. Nix is a purely functional package manager [7]. It allows to easily manage packages and dependencies, and to easily deploy programs on different machines as it guarantees reproducible builds. It is also very useful for development, as it allows to easily create isolated environments for development. This is why we have used Nix for managing the python packages and dependencies. Gradually, Nix has become a superset of other package managers like pip, conda, or cargo.

Any Nix project comes with either a *shell.nix* or a more modern *flake.nix*. Those files are used to describe the project, and to list all necessary dependencies. Since we are developping on NixOS, the integration of Nix with the operating system is very good, and can be easily setup.

Nix is however really straightforward to install on any other distribution through the use of a single script available online. It can be install in as little as one command.

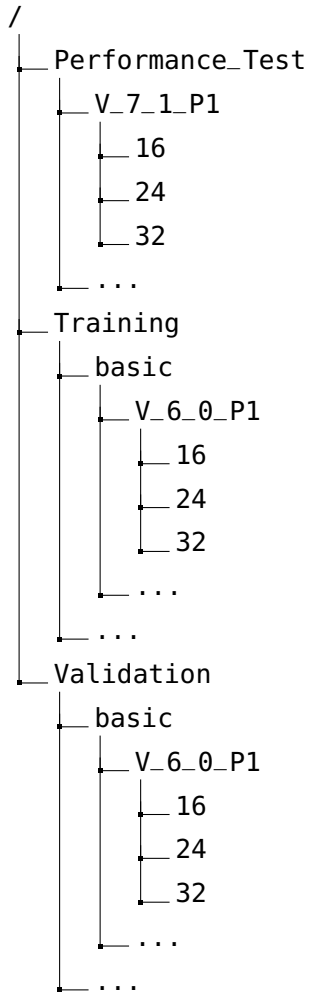
2.2 OpenSSH memory dumps dataset

SmartKex has contributed to the research community by generating a comprehensive annotated dataset of OpenSSH heap memory dumps [6]. The dataset is publicly available on Zenodo [1].

[1] `https://zenodo.org/record/6537904`

The dataset is organized into two top-level directories: $Training$ and $Validation$ with an additional $Performance\_Test$. The first two main directories are further divided based on the SSH scenario, such as immediate exit, port-forward, secure copy, and shared connection. Each of these subdirectories is then categorized by the software version that generated the memory dump. Within these, the heaps are organized based on their key lengths, providing a multi-layered structure that aids in specific research queries.

Figure 2.1: Illustration of the Dataset Directory Structure



Two primary file formats are used to store the data: JSON and RAW. The JSON files contain meta-information like the encryption method, virtual memory address of the key, and the key's value in hexadecimal representation. The RAW files, on the other hand, contain the actual heap dump of the OpenSSH process.

Here is an example of content of a RAW memory dump file, displayed using *vim* and *xdd* commands:

```
00000000: 0000 0000 0000 0000 5102 0000 0000 0000 ........Q.......
00000010: 0607 0707 0707 0303 0200 0006 0401 0206 ................
00000020: 0200 0001 0100 0107 0604 0100 0000 0203 ................
00000030: 0103 0101 0000 0000 0000 0000 0000 0002 ................
00000040: 0001 0000 0000 0000 0000 0100 0000 0001 ................
00000050: 8022 1a3a 3456 0000 007f 1a3a 3456 0000 ."..:4V.....:4V..
00000060: f040 1a3a 3456 0000 9032 1a3a 3456 0000 .@.:4V...2.:4V..
00000070: 608b 1a3a 3456 0000 9047 1a3a 3456 0000 `..:4V...G.:4V..
```

Listing 2.1: 16 bytes per line visualization of a Hex Dump from *Training/basic/V_7_8_P1/16/5070-1643978841-heap.raw*

The original file contains the raw byte content of the heap dump of a specific version of OpenSSH. It is a binary file, which means that it is not human-readable. However, it can be converted to a human-readable format using the *xxd* command. The first column to the left represents the offset in hexadecimal. The last column represents the actual content of the bytes, in ASCII format. The columns in between represent the content of the bytes in hexadecimal format.

Since hexadecimal is a base-16 number system, each byte is represented by two hexadecimal digits. The ASCII representation of the bytes is displayed on the right, and is only used for reference, as it is not always possible to convert the bytes to ASCII. For instance, the bytes at offset 0x10 are not printable characters, and thus cannot be converted to ASCII. Each line represents 16 bytes, and the offset is incremented by 16 for each line.

For the purpose of this thesis, it will be more interesting to visualize the content of the heap dump as 8 bytes lines. This can be achieved by using the *xxd* command with the *-c* option, as shown in the following example:

The same example as before, a memory dump file, displayed using *vim* and *xdd -c 8* commands:

```
00000000: 0000 0000 0000 0000 ........
00000008: 5102 0000 0000 0000 Q.......
00000010: 0607 0707 0707 0303 ........
00000018: 0200 0006 0401 0206 ........
00000020: 0200 0001 0100 0107 ........
00000028: 0604 0100 0000 0203 ........
00000030: 0103 0101 0000 0000 ........
00000038: 0000 0000 0000 0002 ........
00000040: 0001 0000 0000 0000 ........
00000048: 0000 0100 0000 0001 ........
00000050: 8022 1a3a 3456 0000 ."..:4V..
00000058: 007f 1a3a 3456 0000 ...:4V..
00000060: f040 1a3a 3456 0000 .@.:4V..
00000068: 9032 1a3a 3456 0000 .2.:4V..
00000070: 608b 1a3a 3456 0000 `..:4V..
00000078: 9047 1a3a 3456 0000 .G.:4V..
```

Listing 2.2: 8 bytes per line visualization of a Hex Dump from *Training/basic/V_7_8_P1/16/5070-1643978841-heap.raw*

This example shows the exact content of the preceding one.

To this RAW file is associated a JSON file, which contains its annotations.

Here is a example of content of a JSON annotation file that comes with the previous RAW file:


```
        {
           "SSH_PID": "5070",
           "SSH_STRUCT_ADDR": "56343a1a4800",
           "session_state_OFFSET": "0",
           "SESSION_STATE_ADDR": "56343a1a8d30",
           "newkeys_OFFSET": "344",
           "NEWKEYS_1_ADDR": "56343a1aaa40",
           "NEWKEYS_2_ADDR": "56343a1aab40",
           "enc_KEY_OFFSET": "0",
           "mac_KEY_OFFSET": "48",
           "name_ENCRYPTION_KEY_OFFSET": "0",
           "ENCRYPTION_KEY_1_NAME_ADDR": "56343a1a9db0",
           "ENCRYPTION_KEY_1_NAME": "aes128-gcm@openssh.com",
           "ENCRYPTION_KEY_2_NAME_ADDR": "56343a1a3fb0",
           "ENCRYPTION_KEY_2_NAME": "aes128-gcm@openssh.com",
           "key_ENCRYPTION_KEY_OFFSET": "32",
           "key_len_ENCRYPTION_KEY_OFFSET": "20",
           "iv_ENCRYPTION_KEY_OFFSET": "40",
           "iv_len_ENCRYPTION_KEY_OFFSET": "24",
           "KEY_A_ADDR": "56343a1a3170",
           "KEY_A_LEN": "12",
           "KEY_A_REAL_LEN": "12",
           "KEY_A": "feb5fd4ef0759b034d69b858",
           "KEY_B_ADDR": "56343a1a33e0",
           "KEY_B_LEN": "12",
           "KEY_B_REAL_LEN": "12",
           "KEY_B": "f50b988297fa19709445c4ee",
           "KEY_C_ADDR": "56343a1aa1b0",
           "KEY_C_LEN": "16",
           "KEY_C_REAL_LEN": "16",
           "KEY_C": "f5b53280e944db0fe196668d877cd4c0",
           "KEY_D_ADDR": "56343a1a4010",
           "KEY_D_LEN": "16",
           "KEY_D_REAL_LEN": "16",
           "KEY_D": "ac4f18a963d9e72c857497b7dc9d088d",
           "KEY_E_ADDR": "56343a1a7d90",
           "KEY_E_LEN": "0",
           "KEY_E_REAL_LEN": "0",
           "KEY_E": "",
           "KEY_F_ADDR": "56343a1a2f60",
           "KEY_F_LEN": "0",
           "KEY_F_REAL_LEN": "0",
           "KEY_F": "",
           "HEAP_START": "56343a198000"
        }
```


Listing 2.3: Complete JSON example, from *Training/basic/V_7_8_P1/16/5070-1643978841.json*

Those annotation files contain the meta-information about the heap dump, such as the encryption method, virtual memory address of the key, and the key's value in hexadecimal representation. Those annotations are invaluable for the development of machine learning models for key extraction.

The dataset is not just limited to SSH key extraction; it also serves as a resource for identifying essential data structures that hold sensitive information. This makes it a versatile tool for various research applications, including but not limited to machine learning models for key extraction and

malware detection.

2.2.1 Assumptions

Before we dive in, let's make some assumptions about the dataset. We will use these assumptions to guide our exploration of the heap dump file.

- **Heap dump file size:** We will assume that the heap dump file size is a multiple of 8 bytes. This is because the heap dump file is a memory dump, and memory is allocated in chunks that are multiples of 8 bytes. This means that we can expect the heap dump file to be composed of a sequence of 8 bytes blocks. If this assumption is not met, we will assume that padding the last block with 0s will not change the results of our exploration.
- **Chunk chaining:** We will assume that all the heap dump files have been generated using the same `malloc` implementation from GlibC. It means that we can expect to find the same patterns in all the heap dump files. Especially, we expect all the heap dump files to start by a first allocated in-use chunk. We can then follow the malloc header chaining to explore the heap dump file allocated memory chunks [3].
- **Dataset key annotation format:** We will assume that the JSON annotation files have been generated using the same program. This means that we can expect the same format for all the JSON annotation files. This is important, as we will use the JSON annotation files to get the key addresses for annotating memory graphs used for the embedding step. If the format is not the same, we will assume that the JSON annotation file is corrupted, and we will skip it.

The **chunk chaining assumption** is absolutely crucial for the exploration of the heap dump file. It allows us to follow the malloc header chaining to explore the heap dump file allocated memory chunks. This assumption is supported by the code where we can find a comment stating that: «since chunks are adjacent to each other in memory, if you know the address of the first chunk (lowest address) in a heap, you can iterate through all the chunks in the heap by using the size information, but only by increasing address, although it may be difficult to detect when you've hit the last chunk in the heap» [2].

In the scripts and programs that have been developped for the following thesis, we have implemented a number of checks and tests to ensure that these assumptions are met. If not, the programs will raise an error, log the problem and generally skip the data. This behavior is implemented to ensure that the programs are robust to unexpected data, and to ensure that the results are reliable. These assumptions, and related problems will be discussed and measured at several locations in the following sections.

2.2.2 Dataset production system information

Neither the paper „SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump“ nor the dataset itself provide information about the hardware and software used for its generation. This is important since we will be exploring allocated raw bytes which depend on the system and C library used. We obtained some information about the dataset generation by contacting the authors of the paper.

As specified in a mail from Reiser, Hans, the dataset was generated on a system with the following command output:

```
root@debian10:~# ldd --version
ldd (Debian GLIBC 2.28-10) 2.28
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Written by Roland McGrath and Ulrich Drepper.
```

Listing 2.4: Command and logs of the C-library version used for the dataset generation

```
root@debian10:~# lsb_release -a
No LSB modules are available.
Distributor ID:     Debian
Description:        Debian GNU/Linux 10 (buster)
Release:            10
Codename:           buster
```

Listing 2.5: Command and logs of the Linux Standard Base Release used for the dataset generation

```
root@debian10:~# uname -a
Linux debian10 4.19.0-16-amd64 #1 SMP Debian 4.19.181-1 (2021-03-19) x86_64
    GNU/Linux
```

Listing 2.6: Command and logs of the OS and kernel version used for the dataset generation

He also precise that the CPU used was an Intel Xeon CPU, either a E5-2609 or a E3-1230. From those commands, we can deduce the following crucial system related information:

- **CPU architecture:** `x86_64`
- **OS version:** `Debian GNU/Linux 10 (buster)`
- **Kernel version:** `4.19.0-16-amd64`
- **C library version:** `Debian GLIBC 2.28-10`

2.2.3 Conventions and vocabulary

It's important to define some conventions and vocabulary that will be used in the following sections, since many concepts can be ambiguous depending on the context.

- **memory graph:** A memory graph is our particular directed graph that represents the memory of a heap dump file. The memory graph is the main data structure used for the embedding step.
-
- **block:** In the following, we will refer to a block as a sequence of 8 bytes. This is because the heap dump file is a memory dump, and memory is allocated in chunks that are multiples of 8 bytes. This means that we can expect the heap dump file to be composed of a sequence of 8 bytes blocks.

- **chunk:** A chunk is a sequence of blocks bytes. This concepts directly comes from the `malloc` implementation. At its core, a chunk has a user data body composed of blocks and a malloc header block. A chunk can be in-use or free.

2.2.4 Estimating the dataset balancing for key prediction

First, let's quickly estimate what the dataset is composed about. This will later be used to estimate the balancing of data for our key prediction goal. Some quick linux commands can be used to get a general overview of the dataset.

A first command can quickly give us an idea of the number of files in the dataset:

```
1       find /path/to/dataset -type f | wc -l
```

Listing 2.7: Count all dataset files

Another command can be used to get the total size of the dataset:

```
1       du -sb /path/to/dataset
```

Listing 2.8: Get the total size of the dataset

The first command indicates that the dataset contains 208749 files, which represents, according to second one, a total of 18203592048 bytes, or around 18 Gigabytes.

We could just divide the number of files by the size of the dataset to get an average size of the files. However, this would not be accurate, as we are only interested in the size of the RAW files. Since JSON files are much smaller than RAW files, they would skew the average size of the files. Since we are only considering RAW files, we will use improved commands in order to determine the size of the RAW file only.

The following command can be used to get a better understanding of the dataset, concerning the number of RAW files and their size:

```
1       find /path/to/dataset -type f -name "*.RAW" | wc -l
```

Listing 2.9: Find the number of RAW files in the dataset

And the next one can be used to get the number of bytes of RAW files in the dataset:

```
1       find /path/to/dataset -type f -name "*.raw" -exec du -b {} + \
2           | awk '{s+=$1} END {print s}'
```

Listing 2.10: Find the number of bytes of RAW files in the dataset

Where:

- `find phdtrack_data/ -type f -name "*.raw"` finds all the files in the dataset that have the extension `.raw`.
- `-exec du -b {} + | awk '{s+=$1} END {print s}'` executes the command `du -b` on each file found by the previous command, and sums the size of each file.

Theses commands indicate that the dataset contains 103595 RAW files, which represents a total of 18067001344 bytes, or around 18 Gigabytes. This shows that the vast majority of the data is

contained in RAW files, with JSON files representing less than a percent of the dataset in term of byte size. As such, the average size for every RAW file is around 170 Kilobytes.

Now, considering that a given heap dump file is expecting to have only 6 keys (see **??**), with keys maximal possible size being of 64 bytes, we can estimate that we have at maximum 39780480 or around 40 Megabytes of positively labeled samples. This, considering the total useful size of around 18 Gigabytes, means that our dataset is very imbalanced, with an expected upper-bounded ratio of 0.0022% of positively labeled samples or around $2:1000$.

Considering that, a frontal ML binary classification approach will not work. This is why the present report will discuss feature engineering and graph-based memory representation. The idea is to embed more information to our keys so as to be able to fight effectively the imbalanceness of the raw data.

2.2.5 Dataset validation

The dataset is mearly a collection of heap dump RAW files for different use cases and versions of OpenSSH. Each heap dump file goes along a JSON annotation file that has been generated by the creators of the dataset to provide additional information about the heap dump, and especially encryption keys.

However, it is worth noting that the dataset is not perfect. The use of the dataset for machine learning has revealed some issues. For instance, some of the JSON annotation files are not valid JSON files, and cannot be loaded as such. Some of the JSON annotation files are also not complete, with some crucial information missing. This is a problem, as we will use the JSON annotation files to get the key addresses for annotating memory graphs used for the embedding step. If the format is not the same, we will assume that the JSON annotation file is corrupted, and we will skip it.

This is probably due to the fact that the annotations were generated automatically. For instance, in *Training/basic/V_7_8_P1/16/*, literally the first file of the dataset contains an incomplete annotation file, as some of the keys are missing. This is a limitation of the dataset that should be kept in mind when using it for research purposes.

Here is an example of content of a JSON annotation file with missing keys, and with missing annotations (like address or lenght) for the keys that are present:

```
        {
           "ENCRYPTION_KEY_NAME": "aes128-ctr",
           "ENCRYPTION_KEY_LENGTH": "16",
           "KEY_C": "689e549a80ce4be95d8b742e36a229bf",
           "KEY_D": "76788e66a56d2b61eec294df37422fcb",
           "HEAP_START": "5589d41e0000"
        }
```

Listing 2.11: Missing keys in JSON annotation file *Training/basic/V_6_0_P1/16/24375-1644243522.json*

2.2.5.1 Automatic annotation validation

So as to determine really how much of the dataset can be really for machine learned, we have developped a script that checks the validity of thoses annotations. This script called `check_annotations.py`, is used to verify the quality, completeness, consistency and coherence of the

annotations.

Files are regroup under the following categories:

- **Correct and Complete Files:** Files that have no missing keys, and that have all the keys with correct values.
- **Broken Files:** Files that are not valid JSON files, and cannot be loaded as such.
- **Incorrect Files:** Files that have contradictory information in their annotation file.
- **Missing key Files:** Files that have missing keys in their annotation file. A typical example is a JSON file with `"KEY_E": ""`. This means that the key E is missing, and that the key E address is not present in the annotation file, which is a problem for the machine learning since it means that we cannot label correctly the key E.
- **Incomplete key Files:** Files that have incomplete keys in their annotation file. A typical example is a JSON file with `"KEY_E": "689e549a80ce4be95d8b742e36a229bf"`. This means that the key E is present, but that the key E address is not present in the annotation file, which is a problem for the machine learning since it means that we cannot label correctly the key E.

The script is used to validate each JSON file using the following process:

Algorithm 1 Json Annotation Validation

procedure ValidateJson(json_data)
    Initialize errors = Dictionnary{} ▷ Serve as collection for counted errors
    Initialize mandatory_json_keys = [’HEAP_START’, ’SSH_STRUCT_ADDR’, ’SESSION_STATE_ADDR’]
    Initialize key_names = {}
    Initialize incorrect_keys, missing_keys, incomplete_keys = 0
    for mandatory_json_key **in** mandatory_json_keys **do** ▷ Check if some expected json keys are missing
        if mandatory_json_key **not in** json_data **or not** correct hex address **then**
            errors[mandatory_json_key] = False
        else
            errors[mandatory_json_key] = True
        end if
    end for
    for json_key **in** json_data.keys() **do** ▷ Get all the keys names, like A, B, C, D, E, F
        if json_key.startswith("KEY_") **then**
            key_name = GetLetterOfSSHKeyFromJSONKeyName(json_key)
            key_names.add(key_name)
        end if
    end for
    for key_letter **in** key_names **do**
        base_key = "KEY_" + key_letter
        PerformSSHKeyAnnotationValidationAndCompleteness(base_key, json_data) that counts incorrect_keys, missing_keys, incomplete_keys
    end for
    Store counters in errors
    return errors
end procedure

The counting error algorithm done on each SSH key annotation by is described in the following:

Algorithm 2 SSH Key Annotation Validation

```
1: procedure PerformSSHKeyAnnotationValidationAndCompleteness(base_key, json_data)
2:     Initialize incorrect_keys, missing_keys, incomplete_keys = 0
3:     if length(json_data[base_key]) == 0 then
4:         missing_keys += 1                                              ▷ missing key
5:     else
6:         is_key_len_present = exists(json_data[base_key_LEN])
7:         is_key_addr_present = exists(json_data[base_key_ADDR])
8:         is_key_real_len_present = exists(json_data[base_key_REAL_LEN])
9:         if not is_key_len_present or not is_key_addr_present or not is_key_real_len_present then
10:            incomplete_keys += 1                                        ▷ Incomplete keys
11:            Generate and store error message about missing annotations
12:        else if not is_hex_address_correct(json_data[base_key_ADDR]) then
13:            incorrect_keys += 1                                         ▷ Incorrect address
14:            Generate and store error message about incorrect address
15:        else if json_data[base_key_LEN] is not a number or is negative then
16:            incorrect_keys += 1                                         ▷ Incorrect length
17:            Generate and store error message about incorrect length
18:        else
19:            Validate key value length based on annotation length
20:            if json_data[base_key_LEN] == 0 then
21:                missing_keys += 1                                       ▷ missing key
22:            else if length(json_data[base_key]) != json_data[base_key_LEN] * 2 then
23:                incorrect_keys += 1                                     ▷ contradictory length
24:                Generate and store error message about incorrect key value length
25:            end if
26:        end if
27:    end if
28:    return incorrect_keys, missing_keys, incomplete_keys
29: end procedure
```

Note that I have simplified this algorithm. The `is_hex_address_correct` function requires other manipulations to be called, since it checks that the given value is in the range of the heap dump addresses. To do so, it requires handling potentially missing `HEAP_START` annotation, hexadecimal conversion with correct endianness, and other manipulations like determining the size of the heap dump. The full code is available in the `check_annotations.py` file.

The script runs in a few seconds on all the 103595 JSON annotation files, and give the following results:

- **Number of Correct and Complete Files:** 26196 files
- **Number of Broken Files:** 6 files are broken. A direct look at those files shows that they are empty files.
- **Number of Incorrect Files:** 0 files
- **Number of Missing key Files:** 58643 files have missing keys.
- **Number of Incomplete key Files:** 18750 files have incomplete keys.

We can also directly look at the the keys in general:

- **Number of SSH keys:** 546534 keys
- **Number of missing (empty) SSH keys:** 157244 keys
- **Number of incompletly annotated SSH keys:** 37500 keys
- **Number of incorrectly annotated SSH keys:** 0 keys

2.2.6 Dataset cleaning

We need to ensure that the subset of the original dataset that will be used for machine learning is correct and consistent. This means that we need to remove the broken files, and the files that have missing or incomplete keys.

In the new cleaned subset of the dataset, we kept only the files identified as correct and complete. This way, we are left with 26196 RAW files.

From this, we need to remove the raw files that do not respect the **Chunk chaining assumption** 2.2.1. This cleaning process involves the chunk chaining algorithm that will be introduced later. During this process, out of the 26196 RAW files, 5 of them have been detected to have 0 sized chunks. Those files have been removed from the cleaned dataset. This leaves us with 26191 RAW files.

```
1  $ find ~/code/phdtrack/phdtrack_data_clean/ -type f -name "*-heap.raw" | wc -l
2  26191
3  $ find ~/code/phdtrack/phdtrack_data_clean/ -type f -name "*.json" | wc -l
4  26191
```

Listing 2.12: Command and logs of counting the number of RAW files in the cleaned dataset.

In total, this means that only 25.3% of the RAW files with their JSON files are actually usable (correct, complete, with valid chunk chaining), and can be used for machine learning. This is because we don't have access to the packets that have been used to generate the dataset, and thus we cannot regenerate the annotations. Since the machine learning relies entirely on those annotation, we cannot afford to use partially annotated files.

This is a limitation of the dataset that should be kept in mind when using it for research purposes, and especially for supervised machine learning.

2.2.7 Exploring patterns in RAW heap dump files

Before diving into programming, we need to gain a better understanding of how to retreive useful information from heap dump raw file. For that matter, we will continue to experiment with simple commands in RAW heap dump files. Note that in the following, number bases are indicated, since endianness and conversions can get confusing.

Let's start by looking back at the RAW file we already presented in 2.2.

2.2.7.1 Detecting potential pointers

The paper „SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump“ indicates that the keys are 8-bytes aligned. In fact, this is the case for the whole heap dump file. This is why we have choosen to split the study of heap dump files in chunks or blocks of 8 bytes. The term *block* in code is always refering to this, unless specified otherwise. The precision is important, since these blocks should not be confused with *memory blocks* like the ones that are allocated by the `malloc()` function in C.

Let’s re-open the heap dump file in vim, and let’s use the following vim commands to explore the example heap dump file:

- `:%xxd -c 8 5070-1643978841-heap.raw!`: This vim command converts the opened file to a hex dump. The `-c 8` option indicates that we want to display 8 bytes per line.
- `:set hlsearch`: This vim command highlights the search results.
- `:%s/\s\+//g`: This vim command removes all the whitespaces in the file.
- `:%s/\v([0-9a-f]{8}:)/\1\` This vim command adds a whitespace after each 8 byte addresses.
- `:%s/\v(([0-9a-f]{8}: )([0-9a-f]{16}))/\1\` This vim command adds a whitespace after each heap dump byte line.

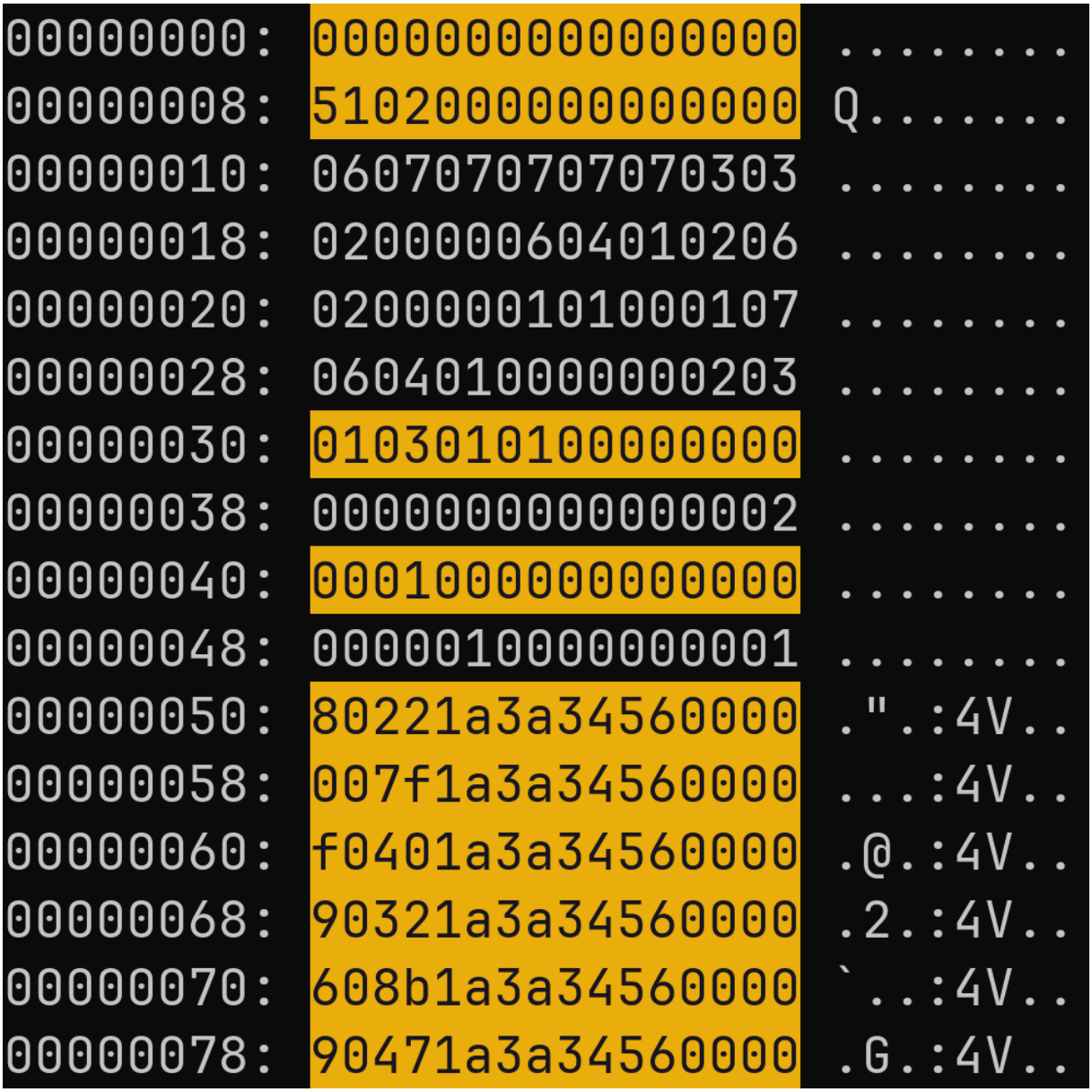
To find potential pointers, we can use the following command in vim:

```
1       :/[0-9a-f]\{12}0\{4}
```

Listing 2.13: Vim command to find potential pointers

This is a search that looks for 12 hexadecimal digits followed by 4 zeros. This is because, the maximum possible addresses in the heap dump file have a size of around 12 hexadecimal digits, and because pointer addresses are in little-endian format, meaning that the last 4 bytes of the address are also the Most Significant Bytes (MSB) of the address.

The result is illustrated below:



We have information about the starting address of the heap using `"HEAP_START": "56343a198000"`. Considering that the example heap dump file contains 135169 bytes, this means that for this given heap dump file, the pointer addresses range from value 94782313037824_{10} and 94782313172993_{10}. Note that the little-endian hexadecimal representation of the heap end address is `0x01901b3a3456` which is 12 character long, or 6 bytes long.

Note that conversions here can get confusing, since potential pointer strings extracted from the heap dump file are given in little-endian hexadecimal format, but the heap start address from the JSON annotation file is given in big-endian hexadecimal format.

That way, we can refine the detection of potential pointers by only considering the bytes that are in the range of the heap. Potential pointers are highlighted with "«<" in the following hex dump:

```
        # conversion from hex to decimal
        def hex_str_to_int(hex_str: str) -> int:
            """
            Convert a normal (big-endian) hex string to an int.
            WARNING: HEAP_START in JSON is big-endian.
            """
            bytes_from_str = bytes.fromhex(hex_str)
            return int.from_bytes(
                bytes_from_str, byteorder='big', signed=False
            )

        def pointer_str_to_int(hex_str: str) -> int:
            """
            Convert a pointer hex string to an int.
            WARNING: Pointer hex strings are little-endian.
            """
            bytes_from_str = bytes.fromhex(hex_str)
            return int.from_bytes(
                bytes_from_str, byteorder='little', signed=False
            )
```

Listing 2.14: Conversions function from hex strings to decimal int values.

Using the functions above, we can check which potential pointers are indeed within the heap dump range.

That way, we can refine the detection of potential pointers. In the following, pointers are highlighted with <<< in the following hex dump:

```
 00000000: 0000000000000000 ........
 00000008: 5102000000000000 Q.......
 00000010: 0607070707070303 ........
 00000018: 0200000604010206 ........
 00000020: 0200000101000107 ........
 00000028: 0604010000000203 ........
 00000030: 0103010100000000 ........
 00000038: 0000000000000002 ........
 00000040: 0001000000000000 ........
 00000048: 0000010000000001 ........
 00000050: 80221a3a34560000 ."..:4V.. <<<
 00000058: 007f1a3a34560000 ...:4V..
 00000060: f0401a3a34560000 .@.:4V.. <<<
 00000068: 90321a3a34560000 .2.:4V.. <<<
 00000070: 608b1a3a34560000 `..:4V.. <<<
 00000078: 90471a3a34560000 .G.:4V.. <<<
```

Listing 2.15: 8 bytes per line visualization of a Hex Dump from *Training/basic/V_7_8_-P1/16/5070-1643978841-heap.raw*

One last check we can do, is verify if the potential pointers are 8-bytes aligned. This can be done by checking if the last 3 bits of the potential address are 0, or using a modulo 8 operation. A simple python function can be used to check that:

```python
def is_pointer_aligned(pointer: int) -> bool:
    """
    Check if a pointer is 8-bytes aligned.
    """
    return pointer % 8 == 0
```

Listing 2.16: Python function to check if a potential pointer is 8-bytes aligned

Using this function on the potential pointers we have found so far, we can see that all of them are indeed 8-bytes aligned. This is a good sign for pointer detection, as we now have a range of tests that can be used to detect potential pointers from other potentially random values.

Here is the pseudo-code for the pointer detection algorithm. This algorithm is presented for a full heap dump file:

Algorithm 3 Pointer Detection Algorithm

```
procedure PointerDetection(heapDumpFile, HEAP_START)
    heapStart ← hex_str_to_int(HEAP_START)
    heapEnd ← heapStart + FileSize(heapDumpFile)
    position ← 0
    potentialPointers ← []
    while position < FileSize(heapDumpFile) do
        block ← Read8Bytes(heapDumpFile, position)
        if block ≠ 0 then
            pointer ← pointer_str_to_int(block)
            if heapStart ≤ pointer ≤ heapEnd then
                if is_pointer_aligned(pointer) then
                    Append(pointer, potentialPointers)
                end if
            end if
        end if
        position ← position + 8
    end while
    return potentialPointers
end procedure
```

This pseudo-code outlines the steps for detecting potential pointers in the heap dump file. It starts by reading the heap dump file 8 bytes at a time. For each 8-byte block, it checks if the block is non-zero and within the heap range. If so, it checks if the potential pointer is 8-bytes aligned using the `is_pointer_aligned` function we described before. If all conditions are met, the potential pointer is added to the list of potential pointers. The algorithm returns this list at the end.

2.2.7.2 Detecting potential keys

Encryption key prediction is the main focus of the present thesis. As such, we will now focus on how to detect potential keys in heap dump files. The paper „SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump“ introduces 2 algorithms for key detection. The first one is a brute force approach that consists in trying all the possible keys in the heap dump file.

The second one is a more sophisticated approach that uses a set of rules to detect potential keys.

The first brute-force algorithm is given by:

Algorithm 4 SSH keys brute-force algorithm from „SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump“ [6]

```
1: procedure FindIVAndKey(netPacket, heapDump)
2:     ivLen ← 16                                   ▷ Based on the encryption method
3:     keyLen ← 24                                  ▷ Based on the encryption method
4:     i ← sizeof(cleanHeapDump)
5:     r ← 0
6:     while r < i do
7:         pIV ← heapDump[r : r + ivLen]
8:         x ← 0
9:         while x < i do
10:            pKey ← heapDump[x : x + keyLen]
11:            f ← decrypt(netPacket, pIV, pKey)
12:            if f is TRUE then
13:                return pIV, pKey
14:            end if
15:            x ← x + 8                            ▷ The IV is 8-bytes aligned
16:        end while
17:        r ← x + 8                                ▷ The key is 8-bytes aligned
18:    end while
19: end procedure
```

Algorithm 1 outlines the brute-force approach for finding the Initialization Vector (IV) and the key. Initially, the lengths ivLen and keyLen are set based on the encryption method used for the heap. The algorithm then takes the first ivLen bytes from the heap dump to serve as the potential IV (pIV). Subsequently, keyLen bytes are extracted from the heap dump, starting from the first byte, to act as the potential key ($pKey$). The algorithm iterates through this potential key until it reaches the end of the heap dump. If decryption of the network packet is unsuccessful, the process is repeated by reading the next potential IV and the subsequent potential key [6].

This algorithm is fairly straightforward, and can be implemented in a few lines of code. However, it is also very inefficient, as it tries all the possible keys in the heap dump file. It also needs some encrypted network packets to be able to test the keys, which are not included in the dataset. As such, we will not implement this algorithm.

This is why the authors of the paper have also developed a more sophisticated algorithm that uses a set of rules to detect potential keys.

No pseudo-code is given for the second algorithm, but the paper „SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump“ gives a description of the algorithm. It rely on the high-entropy assumption of encryption keys. The algorithm is inspired by image processing techniques, and can be described as follows:

This Preprocessing Algorithm serves as a crucial step in adapting the heap data for machine learning models. The algorithm begins by reshaping the raw heap data into an $N \times 8$ matrix X, since keys are 8-bytes aligned [6]. Here, $N \times 8$ is the size of the original heap data in bytes. It then calculates the discrete differences of the bytes in both vertical and horizontal directions, storing the results in matrix Y. The algorithm employs a logical AND operation on these differences to

Algorithm 5 Image-processing inspired Preprocessing Algorithm, as described in „SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump“ [6]

procedure Preprocessing(heapData)
    Reshape heapData into $N \times 8$ matrix X
    for $i = 0$ to $N - 1$ **do**
        for $j = 0$ to 7 **do**
            $Y[i][j] = |X[i][j] - X[i][j+1]| \& |X[i][j] - X[i+1][j]|$
            $Z[i] = \text{count}(Y[i][k] == 0) \geq 4$
            if $i < N - 1$ **then**
                $R[i] = Z[i] \& Z[i+1]$
            end if
        end for
    end for
    Extract 128-byte slices from R for training
end procedure

identify high-entropy regions, which are likely candidates for encryption keys. Each 8-byte row in Y is examined for randomness, and if at least half of its bytes differ from adjacent bytes, it is marked as a potential part of an encryption key. The algorithm then filters out isolated rows that are unlikely to be part of an encryption key, resulting in an array R. Finally, 128-byte slices are extracted from R for training the machine learning model. This preprocessing step not only adapts the data for machine learning but also narrows down the search space for potential encryption keys, thereby enhancing the efficiency of the subsequent steps.

However, this algorithm is not as efficient as it could be. It rely on using a kind of sliding window, which is not easily parallelizable. Also, the entropy-inspired computation is not as straightforward as it could be. That why we propose a new algorithm that is more efficient and more easily parallelizable.

In order to perform some ML techniques, and because the keys we are looking for can have a range of possible lengths (16, 24, 32, or 64 bytes), we shift the focus from detecting the full key, to just be able to predict the address of the key. That way, we can deal with keys of different sizes, and we can also use the same algorithm to detect the IV. This is why we will focus on detecting potential keys addresses, and not the full keys.

We thus introduce a new algoritm for narrowing the search space for potential keys. This algorithm is inspired by the paper „SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump“, but is more efficient and more easily parallelizable, as it focuses on producing pairs of blocks of 8 bytes with high entropy. It uses directly the Shannon entropy formula, with each entropy computation being independent from the others.

Algorithm 6 Entropy Based Detection of Potential Key blocks

procedure EntropyDetection(heapData)
    Pad heapData with 0s to be 8-bytes aligned
    Reshape heapData into $N \times 8$ matrix X
    for $i = 0$ to $N - 1$, **do**
        $entropy[i] = \text{ShannonEntropy}(X[i])$ ▷ Independents, compute in parallel.
    end for
    Add $entropy$ 2 by 2 pairs into $entropy\_pairs$ ▷ Keep block indexes in resulting tuples.
    Sort $entropy\_pairs$ by entropy as $sorted\_pairs$
    return SortedPairs(sorted_pairs)
end procedure

The *Entropy Based Detection of Potential Key blocks* algorithm takes a raw heap dump, represented by the variable `heapData`, as input. The data is first padded with zeros to align it to 8-byte blocks and then reshaped into an $N \times 8$ matrix X. The Shannon entropy is computed for each 8-byte block in parallel, resulting in an array `entropy`. Subsequently, the entropy values of adjacent blocks are summed to form pairs, which are stored in `entropy_pairs` along with their block indexes. These pairs are then sorted by their entropy sums to produce `sorted_pairs`. The idea of using pairs of blocks instead of a single block or more that two blocks is related to the minimim key size, which is 16 bytes. This means that we need at least 2 blocks to be able to detect a potential key. The algorithm returns sorted pairs, so that we can easily extract the ones with the highest entropy sums. Given the index of a block, its actual memory address can be recomputed using the `HEAP_START` address available in annotations.

Using this algorithm, let's continue to explore our example heap dump file from 2.2. We will use the following python function to compute the Shannon entropy of a given block of 8 bytes:

```
        def get_entropy(data: bytes):
            """
            Computes the entropy of a byte array, using Shannon's formula.
            """

            if len(data) == 0:
                return 0.0

            # Count the occurrences of each byte value
            _, counts = np.unique(data, return_counts=True)

            # Calculate the probabilities
            prob = counts / len(data)

            # Calculate the entropy using Shannon's formula
            entropy = -np.sum(prob * np.log2(prob))

            return entropy
```

Listing 2.17: Python function to compute the Shannon entropy of a given block of 8 bytes

This function used np array function for efficient computation. We can now use this function to compute the entropy of each block of 8 bytes in the heap dump file. We can then sort the pair of blocks by their entropy, and keep the ones with the highest entropy.

When applied to the file *Training/basic/V_7_8_P1/16/5070-1643978841-heap.raw*, the algorithm produced 16896 entropy pairs, with 631 pairs having the maximum entropy sum. Another test using the index to address conversion and the JSON annotation file also indicate that all of the 6 key addresses are within the 631 pairs with the highest entropy sum.

This allows to reduce significantly the search space for potential keys, to already less that 4% of the original heap dump file, which is significantly better that the 30% reduction obtained by the preprocessing algorithm described in the paper SmartKex [6], but less that the 2% reduction obtained by the ML-based processing algorithm described in the paper [6]. However the same paper indicated that it was tested only for Key A and Key C, whereas this algorithm is tested for all the keys. Keep in mind that this is just an example for a single random file in the dataset, as a way to introduce the subject. In-depth experiments will be done in the dedicated chapter on Machine Learning.

Indeed, it is important to mention that we can rely on the JSON annotation files for providing

labelling for key address prediction. Using this, we do not need to decrypt the network packets to be able to train our ML models. This is a huge advantage, and is also required since we don't have the encrypted network packets in the dataset. Since we don't have those, and since the keys are already given, that is why we will focus on key address prediction, and not on key prediction.

2.2.8 Data structure exploration

Since the dataset contain whole heap dump file, we can also try to detect allocated chunks in those heap dumps. This can be done by looking for patterns in the heap dump file, in a similar fashion as we have done for potential pointers. However for data structure, we can rely on our knowledge of the C library used to know exactly what to look for.

Since OpenSSH is written in C, we can expect to find some C memory chunks in the heap dump files. C uses the `malloc` function to allocate memory. This function is used to allocate memory for a given data structure. It takes as input the size of the data structure to allocate, and returns a pointer to the allocated memory. We know that the dataset has been produced using `GLIBC 2.28` 2.2.2. Looking directly at the code for `malloc` in `GLIBC 2.28`, we can read in the comments that «Minimum overhead per allocated chunk: 4 or 8 bytes. Each malloced chunk has a hidden word of overhead holding size and status information» [2]. This is what we refer to as the *malloc header*. This means that we can expect to find some 8-bytes aligned blocks in the heap dump file, that are not pointers, but that are the result of a `malloc` call. Detecting and using those *malloc headers* is how we will try to detect memory chunks in heap dump files.

In Linux on a `x86_64` architecture, the malloc function typically uses a block (chunk) header to store metadata about each allocated block. This header is placed immediately before the block of memory returned to the user. The exact layout can vary depending on the implementation of the C library (e.g., glibc, musl), but generally, it contains the following:

- **Size of the Block**: The size of the allocated block, usually in bytes. This size often includes the size of the header itself and may be aligned to a multiple of 8 or 16 bytes.
- **Flags**: Various flags that indicate the status of the block, such as whether it is free or allocated, or whether the previous block is free or allocated. These flags are often stored in the least significant bits of the size field, taking advantage of the fact that the size is usually aligned, leaving the least significant bits zeroed.

2.2.8.1 How `malloc` handles Heap Allocation

The `malloc` function in GLIBC 2.28 uses a boundary tag method to manage chunks of memory. Each chunk contains metadata that helps in the allocation and deallocation of memory [2] [3]. Below are the key components of a chunk:

A chunck is a contiguous section of memory, in our case composed of several blocks of 8 bytes, that is handled by `malloc`. It contains the following components [3] [4]:

1. **Size of Previous Chunk**: This field exists only if the previous chunk is unallocated and its `P` (PREV_INUSE) bit is clear. It helps in finding the front of the previous chunk.
2. **Size of Chunk**: This field contains the size of the chunk in bytes along with three flags: `A` (NON_MAIN_ARENA), `M` (IS_MMAPPED), and `P` (PREV_INUSE). These flags are in the last 3

LSBs of the size field. This precise block is considered in the following report as the *malloc header* block. Note that the size of the chunk include the size of the *malloc header*, chunk user data and *footer* blocks.

3. **User Data**: This is the actual memory space that is returned to the user.

4. **Footer**: This is the same as the size of the chunk but is used for application data. Depending on how the chunk is represented, this is exactly the same as the **Size of Chunk** field. This is a more intuitive representation and is the one choosen in the schematical representation below.

5. **Flags**:

    - `A` (NON_MAIN_ARENA): Indicates if the chunk is in the main arena or a thread-specific arena.
    - `M` (IS_MMAPPED): Indicates if the chunk is allocated via `mmap`.
    - `P` (PREV_INUSE): Indicates if the previous chunk is in use. If false, it means the previous chunk is free.

The chunck allocation process involves the following concepts:

1. **Initialization**: The very first chunk allocated always has the `P` bit set to prevent access to non-existent memory.

2. **Free Chunks**: Free chunks are stored in circular doubly-linked lists. They contain forward and backward pointers to the next and previous chunks in the list.

3. **Mmapped Chunks**: These chunks have the `M` bit set in their size fields and are allocated one-by-one.

4. **Fastbins**: These are treated as allocated chunks and are consolidated only in bulk. These *bins* are used to speed up the allocation process.

5. **Top Chunk**: This is a special chunk that always exists. If it becomes less than `MINSIZE` bytes long, it is replenished.

As explained directly in the code comments, an allocated chunck of 8 byte blocks can be described by the diagram below [2]. Note that is representation is personal to better suit the needs of our forensic annalysis. The slight difference resides in the fact that the `footer` with the size of the considered chunk is represented as being part of the next chunk and not the current chunk. The footer of the previous chunk is actually the `mchunkptr` address. As stated in the GlicC wiki: «within the malloc library, a "chunk pointer" or `mchunkptr` does not point to the beginning of the chunk, but to the last word in the previous chunk - i.e. the first field in mchunkptr is not valid unless you know the previous chunk is free» [3]. Due to consideration of free/allocated chunks, it’s actually easier to just consider the footer as being part of the next chunk, and not the current chunk. This is why the diagram below is slightly different from the one in the GLIBC wiki. This is just a difference in schematical representation, and does not change the actual data structure.

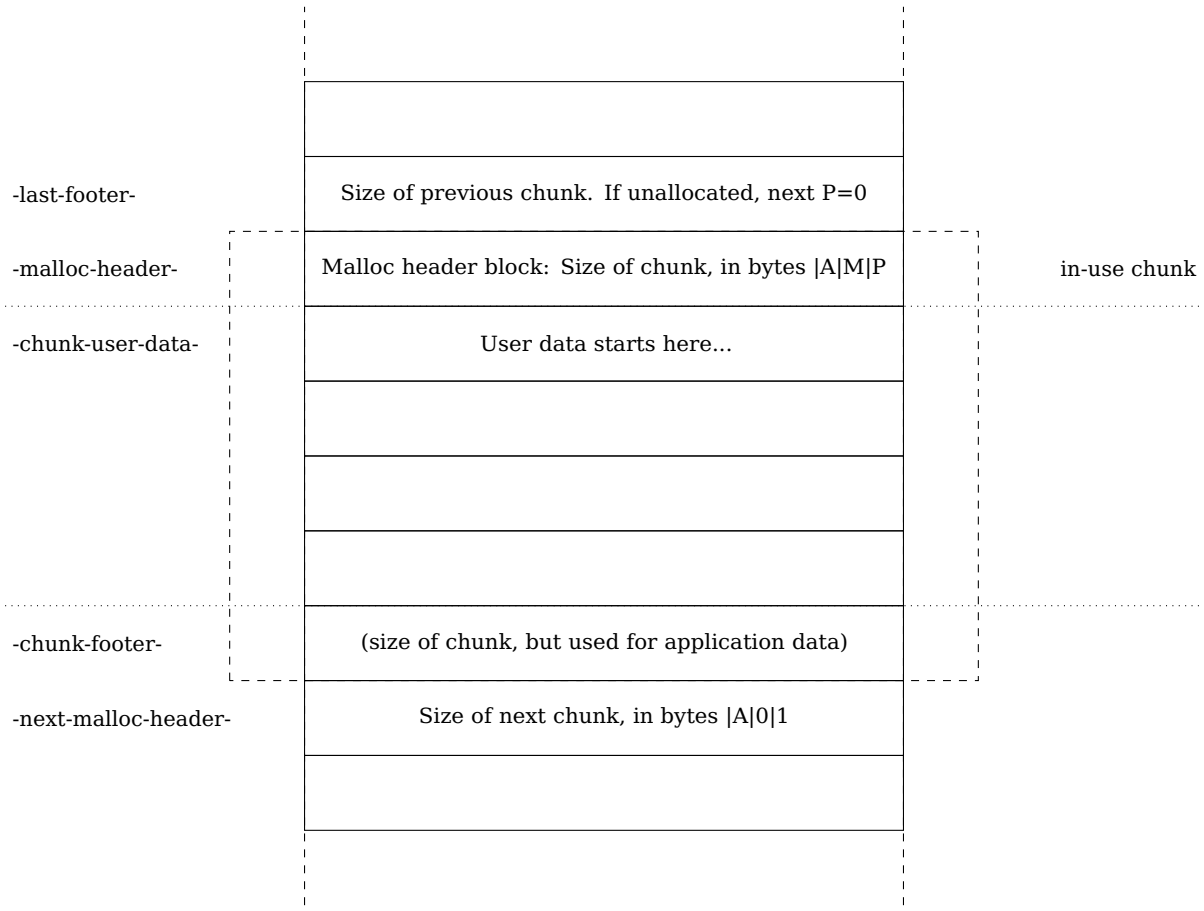


Figure 2.2: Diagram of an allocated chunk in GLIBC 2.28 [2].

The `malloc` function in GLIBC 2.28 uses a boundary tag method to manage chunks of memory. Each chunk contains metadata that helps in the allocation and deallocation of memory [2] [3]. The library stores available free chunks in circular doubly-linked lists called «bins». This allows to quickly find a free chunk of memory of a given size. The problem is that we don't have access to those bins in the heap dump file. So to detect if a given chunk is in-use or free, we can rely on several methods. The first one is to look at the `P` bit of the malloc header. If it is set to 1, it means that the chunk is in use. If it is set to 0, it means that the chunk is free.

I also remarked that sometimes, the given heap dump file is cropped and the last block is only composed of zeros and not complete. This is for instance the case with the last chunk of *Training/basic/V_7_1_P1/24/17016-1643962152-heap.raw*.

```
1 WARN: Chunk [94022266975200] Chunk(block_index=10876, size=48176, flags=[A=False, M=False
      , P=True]) is out of bounds. Last block index: 16895 Iteration index: 16896
2 WARN: Chunk [94022266975200] Chunk(block_index=10876, size=48176, flags=[A=False, M=False
      , P=True]) is out of bounds. Last block index: 16895 Iteration index: 16897
3 Chunk(block_index=10876, size=48176) is only composed of zeros.
```

Listing 2.18: Logs from chunk exploration script, related to the last chunk of the file *Training/basic/V_7_1_P1/24/17016-1643962152-heap.raw*.

A free chunk contains the pointers of the next and previous free chunks in the heap, for its given bin. A representation of a free chunk, based directly on the code documentation [2], is given below:

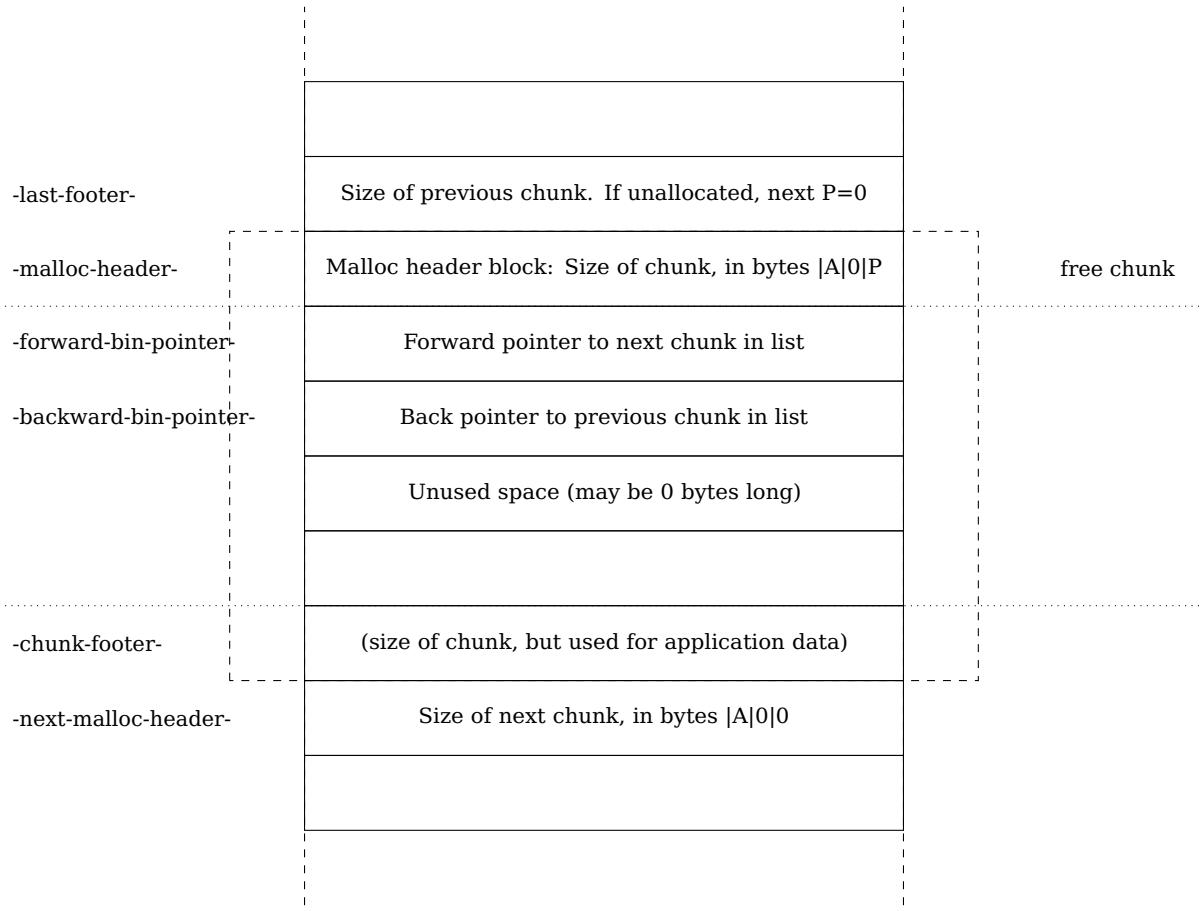


Figure 2.3: Diagram of a free chunk in GLIBC 2.28 [2].

2.2.8.2 Chunk chaining

The chunk chaining algorithm relies on the **chunck chaining assumption** 2.2.1. This assumption states that the allocator allocates chunks after chunks, and that the chunks are contiguous in memory. This means that we can expect to find the malloc header of the next chunk at the address current_malloc_header_chunk_address + current_chunk_size + 8, where 8 is the size of the malloc header block, or current_chunk_user_data_address + current_chunk_size. This is the case for both free and allocated chunks. This is why we can use this assumption to detect chunks in the heap dump file.

This necessitates to understand malloc header blocks, and how they are represented in the heap dump file. In the specific case of `GLIBC 2.28`, the malloc header is defined as follows:

```
1 #define SIZE_BITS (PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)
```

Listing 2.19: Malloc header definition in `GLIBC 2.28`

Since the malloc header respects the endianness of the system, we can expect to find the malloc header in little-endian format in the heap dump file. Using vim on *Training/basic/V_7_-8_P1/16/5070-1643978841-heap.raw*, we can use the following command to find some potential malloc headers:

```
1 :/[0-9a-f]\{4}0\{12}
```

Listing 2.20: Vim command to find potential malloc headers

This gives something like the following:

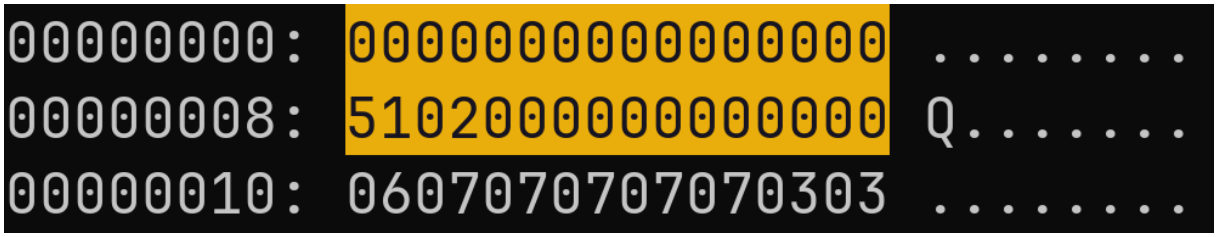


Figure 2.4: Attempt at malloc header detection in *Training/basic/V_7_8_P1/16/5070-1643978841-heap.raw*, at heap start.

Indeed, after a first zero block of 8 bytes (potential previous chunk footer), we expect a first data structure to be allocated at the start of the heap. Here this data structure is of size 5102000000000000_{16LE} (little-endian hex format) or 593_{10} bytes. The fact that it is an odd number is due to the LSB being set to 1, to indicate that the preceding chunk is allocated (P flag). This means that the real size of the structure is actually $593_{10} - 1_{10} = 592_{10}$. This value is 8-byte aligned.

Since we know that the allocator allocates chunks after chunks, we can expect the next chunk to be allocated at the address $5102000000000000_{16LE} + 592_{10} + 8_{10} = 5882193a34560000_{16LE} =$. Note that we need to add 8 to the size to account for the malloc header block.

In vim, since the address start at 0, we have to look at $592_{10} + 8_{10} = 258_{16}$. Let's have a look there:

Figure 2.5: Attempt at malloc header detection in *Training/basic/V_7_8_P1/16/5070-1643978841-heap.raw*, at index $592_{10} = 250_{16}$.

There, we can see a zero block, followed by what we can expect to be another malloc header at address 258_{16}. By doing the same process, we can thus propose an algorithm to detect the malloc headers, and thus the structures in the heap dump file.

First, here is a simple algorithm to extract all the necessary information from a malloc header block:

Algorithm 7 Malloc Header Parsing Algorithm

procedure MallocHeaderParsing($block$)
Require: $block$ is a block of 8 bytes
Ensure: $MallocHeader$ object
Ensure: $Flags$ object
    Note: In this algorithm, $\&$ represents bitwise AND, and $\sim$ represents bitwise negation.
    $size\_and\_flags \leftarrow$ ConvertBytesToInteger($block$, 'little-endian')
    $size \leftarrow size\_and\_flags \ \& \ (\sim 0x07)$ ▷ Clear the last 3 bits to get the size
    $Flags.a \leftarrow$ bool($size\_and\_flags \ \& \ 0x04$)
    $Flags.m \leftarrow$ bool($size\_and\_flags \ \& \ 0x02$)
    $Flags.p \leftarrow$ bool($size\_and\_flags \ \& \ 0x01$)
    return $MallocHeader\{size, Flags\}$
end procedure

We can also isolate the size parsing algorithm into a handy function:

Algorithm 8 Malloc Header block to size conversion Algorithm

procedure ConvertToSize($block$)
Require: $block$ is a block of 8 bytes
    Note: In this algorithm, $\&$ represents bitwise AND, and $\sim$ represents bitwise negation.
    $size\_and\_flags \leftarrow$ ConvertBytesToInteger($block$, 'little-endian')
    $size \leftarrow size\_and\_flags \ \& \ (\sim 0x07)$ ▷ Clear the last 3 bits to get the size
    return $size$
end procedure

Based on those algorithms, and in a similar fashion as what we have done manually by exploring the heap dump file with vim, we can propose the following algorithm to detect the malloc headers in a heap dump file:

Algorithm 9 Malloc Header Chaining Algorithm

procedure MallocHeaderDetection($heapDumpFile$)
    Note: ConvertToSize is equivalent to MallocHeaderParsing($block$).size ▷ See 2.2.8.2
    Initialize malloc_header_list to empty list
    $position \leftarrow 0$
    while $position <$ FileSize($heapDumpFile$) **do**
        $block \leftarrow$ Read8Bytes($heapDumpFile, position$)
        if block $\neq 0$ **then**
            $size \leftarrow$ ConvertToSize($block$) ▷ Be careful with flags
            Assert $size! = 0$
            Assert $size \mod 8 = 0$ ▷ Check if the size is 8-bytes aligned
            $position \leftarrow position + size$ ▷ Leap over data structure.
        else
            $position \leftarrow position + 8$
        end if
    end while
    return $malloc\_header\_list$
end procedure

The idea behing the malloc header detection algorithm is simple. We start at the beginning of

the heap dump file, and we look for the first non-zero block. We then assume that the next block is a malloc header. We convert it to a size, and then leap over the user data and the footer up to the next chunk malloc header block index. The process is repeated until reaching the end of the heap dump file.

Note that in case of a problem, like when the size obtained from malloc header parsing is equal to 0, this means that the heap dump chaining is broken. This has been handled in the dataset cleaning section 2.2.6.

2.2.8.3 Chunk chaining example

The program `chunk_algorithms.py` has been developped specifically to test the chunk parsing and refine the associated algorithms.

We can test our chunk parsing algorithm on a test file in the cleaned dataset.

```
$ python src/data_structure_detection.py --input /home/onyr/code/phdtrack/
    phdtrack_data_clean/Training/Training/basic/V_7_1_P1/24/17016-1643962152-heap.raw --
    debug
    Datetime: 2023_09_27_17_08_23_157209
    Chunk [1]: Chunk(block_index=2, size=592, flags=[A=False, M=False, P=True])
    Chunk [2]: Chunk(block_index=76, size=32, flags=[A=False, M=False, P=True])
    Chunk [3]: Chunk(block_index=80, size=32, flags=[A=False, M=False, P=True])
    Chunk [4]: Chunk(block_index=84, size=32, flags=[A=False, M=False, P=True])
    Chunk [5]: Chunk(block_index=88, size=32, flags=[A=False, M=False, P=True])
    Chunk [6]: Chunk(block_index=92, size=192, flags=[A=False, M=False, P=True])
    Chunk [7]: Chunk(block_index=116, size=32, flags=[A=False, M=False, P=True])
    Chunk [8]: Chunk(block_index=120, size=32, flags=[A=False, M=False, P=True])
    Chunk [...]: ...
    Chunk [911]: Chunk(block_index=10194, size=128, flags=[A=False, M=False, P=True])
    Chunk [912]: Chunk(block_index=10210, size=256, flags=[A=False, M=False, P=True])
    Chunk [913]: Chunk(block_index=10242, size=160, flags=[A=False, M=False, P=True])
    Chunk [914]: Chunk(block_index=10262, size=512, flags=[A=False, M=False, P=True])
    Chunk [915]: Chunk(block_index=10326, size=1296, flags=[A=False, M=False, P=True])
    Chunk [916]: Chunk(block_index=10488, size=1552, flags=[A=False, M=False, P=True])
    Chunk [917]: Chunk(block_index=10682, size=1552, flags=[A=False, M=False, P=True])
    Chunk [918]: Chunk(block_index=10876, size=48176, flags=[A=False, M=False, P=True])
    ----------> Statistics:
    Total number of files: 1
    Total number of chunks: 918
    Total number of blocks: 16896
    Total number of chunks with P=1: 903
    Total number of chunks with M=1: 0
    Total number of chunks with A=1: 0
    Total number of chunks only composed of zeros: 1
```

Listing 2.21: Testing chunk parsing on *Training/basic/V_7_1_P1/24/17016-1643962152-heap.raw*. Partial log output.

Looking at the first allocated chunks, we recognize what we had seen manually with vim for the file *Training/basic/V_7_8_P1/16/5070-1643978841-heap.raw*. The first chunk is of size 592, and the next one is of size 32. This is exactly what we had seen manually. It is a good sign that our algorithm is working as expected. We can also see that the last chunk is of size 48176, which is significantly bigger than the other chunks. This chunk is only composed of zeros, and is truncated, meaning that its size if bigger than the actual size of the heap dump file.

2.2.8.4 Distinguishing between free and allocated chunks

The malloc header chaining algorithm allows to detect memory chunks in the heap dump file. However, it does not allow to distinguish between free and allocated chunks. This is a problem, since we want to be able to distinguish between free and allocated chunks, to be able to detect potential data structures and filter out useless blocks.

Considering the structural differences between a free and in-use block, it's possible to try distinguishing free blocks by their *forward* and *backward* pointers. The issue is that the head dump raw file are not provided with any *bins* information. As such, distinguishing between two normal pointers and the ones expected inside a free block is a non-trivial task. Hence, the tests performed on this idea are inconclusive. A more straightforward technique is to rely on the P malloc header flags.

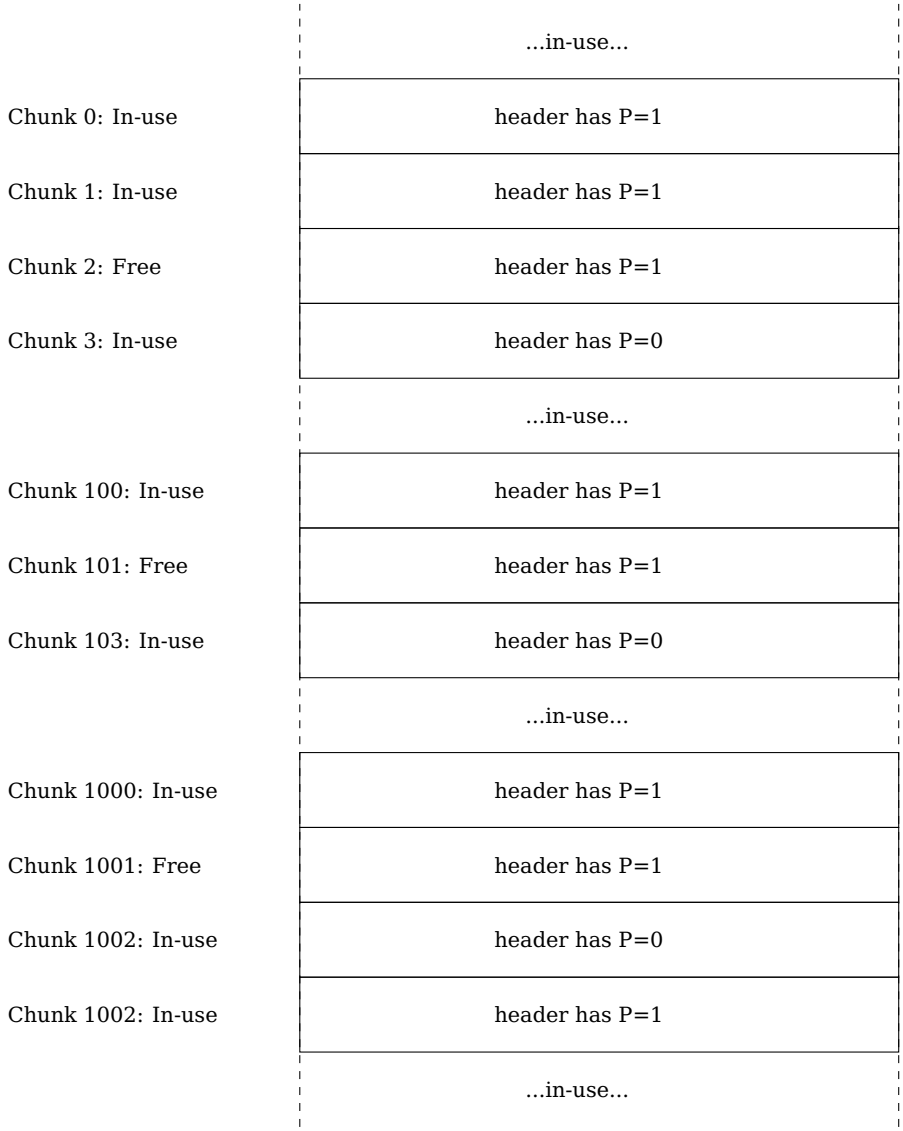


Figure 2.6: Heap dump showing a mix of free and in-use chunks. Note: each chunk immediately after a free chunk has a P flag set to 0. Each rectangle represents a chunk.

For a given chunk, the follow-up chunk in ascending address number, contains such a flag in its header block. If the flag value is 0, then the current chunk is free. If the flag value is 1, then the current chunk is in use by the program. This is the technique that has been used in the final

implementation of the chunk chaining algorithm.

Algorithm 10 Chunk Parsing Algorithm

procedure ChunkParsing($heapDumpFile, HEAP\_START$)
    Note: ConvertToSize is equivalent to MallocHeaderParsing($block$).size
    Note: Get8BytesBlocks returns a list of 8 bytes blocks from the heap dump file.
Ensure: $MallocHeader$ object
Ensure: $Flags$ object
Ensure: $Chunk$ object
Ensure: $HEAP\_START$ provided from annotation file is a correct address.
    Note: In this algorithm, $\&$ represents bitwise AND, and $\sim$ represents bitwise negation.
    Initialize $chunk\_list$ to empty list
    $blocks \leftarrow$ Get8BytesBlocks($heapDumpFile$)
    Initialize $index \leftarrow 0$
    while $index < lenght(blocks)$ **do**
        $block \leftarrow blocks[index]$
        Initialize $Chunk$ to empty object
        if block $\neq 0$ **then**
            $Chunk.header : \{size, Flags\} \leftarrow$ MallocHeaderParsing($block$) ▷ See 2.2.8.2
            Assert $Chunk.header.size \geq 2$ ▷ Must contains at least header and footer
            Assert $Chunk.header.size \mod 8 = 0$ ▷ Check if the size is 8-bytes aligned
            $Chunk.block\_index \leftarrow index$ ▷ Index of the first block of the chunk after header
            $Chunk.address \leftarrow HEAP\_START + (index * 8)$ ▷ Address of $block\_index$
            $footer\_index \leftarrow index + Chunk.header.size - 1$ ▷ Index of the footer block
            if $footer\_index < lenght(blocks)$ **then**
                $footer \leftarrow blocks[footer\_index]$
                if ConvertToSize($footer$) $= Chunk.footer.size$ **then**
                    $Chunk.correct_footer \leftarrow True$
                else
                    $Chunk.correct_footer \leftarrow False$
                end if
            else
                $Chunk.correct_footer \leftarrow False$
            end if
            $next\_chunk\_header\_index \leftarrow index + Chunk.header.size$ ▷ Index of the next chunk header block
            if $next\_chunk\_header\_index < lenght(blocks)$ **then**
                $next\_chunk\_header \leftarrow blocks[next\_chunk\_header\_index]$
                $Chunk.is\_in\_use \leftarrow$ MallocHeaderParsing($next\_chunk\_header$).$flags.p$
            else
                $Chunk.is\_in\_use \leftarrow False$ ▷ See [2]
            end if
            $index \leftarrow index + Chunk.header.size$ ▷ Leap over chunk.
        else
            $index \leftarrow index + 8$ ▷ Leap over zero block.
        end if
    end while
    return $malloc\_header\_list$
end procedure

Note that this algorithm is based on the malloc header chaining algorithm 2.2.8.2. The main difference is that we now have access to the malloc header flags from the following chunk, and that we can thus distinguish between free and allocated chunks. The algorithm also includes the footer parsing technique discussed briefly in the following section.

2.2.8.5 Chunk footer

The documentation of the `malloc` function of GLIBC states that the footer of a chunk is the same as the size of the chunk considered. In the current report, we represent the footer as being part of the chunk itself.

Below are two chunks content of similar size:

```
Printing Chunk [addr:0x80a2d1438355] [status:in-use] [footer:incorrect] Chunk(block_index
    =80, size=32, flags=[A=False, M=False, P=True])
        Block [79]:     b'!\x00\x00\x00\x00\x00\x00\x00'        33      -malloc-header-
        Block [80]:     b'\xa0\xa2\xd1C\x83U\x00\x00'       94022266888864
        Block [81]:     b'\xc0\xa2\xd1C\x83U\x00\x00'       94022266888896
        Block [82]:     b'\x00\x00\x00\x00\x00\x00\x00\x00'         0       -footer-
Printing Chunk [addr:0xa09fd2438355] [status:free] [footer:correct] Chunk(block_index
    =8180, size=32, flags=[A=False, M=False, P=True])
        Block [8179]:   b'!\x00\x00\x00\x00\x00\x00\x00'        33      -malloc-header-
        Block [8180]:   b'\xb0\xbc\xe1\xeeS\x7f\x00\x00'        139998466784432
        Block [8181]:   b'\xb0\xbc\xe1\xeeS\x7f\x00\x00'        139998466784432
        Block [8182]:   b' \x00\x00\x00\x00\x00\x00\x00'        32      -footer-
```

Listing 2.22: Printing some free and in-use chunks from *Training/basic/V_7_1_P1/24/17016-1643962152-heap.raw*.

Here, the status of the chunk has been detected using the `P` flag technique. At first sight, those two blocks seems similar. The first 2 blocks in the user data space of the chunks both seems to contain what look like pointers. As one can see, the first chunk in this example, with a `block_index=80` has clearly a malloc header and footer as expected. Note that here, the value 33 represents the size of the block (32 bytes which correspond to 4 blocks) with the LSB being set to 1 meaning the preceding chunk is in use. However, the in-use block footer doesn't correspond to the value we expect. This difference of behavior is observed throughout the cleaned dataset.

2.2.9 Discussing chunk parsing for problem scale reduction

Now that we have presented all the necessary knowledge and algorithms used to be able to parse the dataset, we can discuss the results of those algorithms and their uses and limitations. Many tests have been needed in order to develop the final algorithms. This testing process has also unveiled some interesting properties of the dataset that will be used as basis for the semantic embedding of the memory graph representation and subsequent machine learning steps.

The program `chunk_algorithms.py` has been developped specifically to test the chunk parsing and refine the associated algorithms on the cleaned dataset. Below are presented the global statistics produced by the final version of the program:

```
        Input is directory: /home/onyr/code/phdtrack/phdtrack_data_clean/
        Found 26191 files in /home/onyr/code/phdtrack/phdtrack_data_clean/.
        Processing files: 100%|\blacksquare \blacksquare \blacksquare \blacksquare \
            blacksquare | 26191/26191 [12:11<00:00, 35.81it/s, file=7091-1650972335]
```

```
        ------> Statistics:
        Total number of parsed files: 26191
        Total number of skipped files: 0
        Total number of chunks: 37682063
        Total number of blocks: 674232832
        Total number of chunks with P=1: 37346373
        Total number of chunks with M=1: 0
        Total number of chunks with A=1: 0
        Total number of free chunks: 354410
        Total number of chunks only composed of zeros: 18720
        Total number of blocks in free chunks: 183331224
        Total number of chunks with correct footer value: 1009522
        Total number of chunks both free and with correct footer value: 335690
        Total number of chunks free and annotated: 0
        Total number of potential footers with annotations (should be 0): 0
        Total number of annotated chunks: 209528
        Total number of chunks in used, with correct footer, and annotated: 7668
        Total number of chunks in used, with correct footer, and key annotated: 7668
        Percentage of free chunks: 0.9405270619074121%
        Percentage of blocks in free chunks: 27.19108523033183%
        Percentage of free chunks with correct footer value: 94.71798199825061%
        Percentage of in-use chunks with correct footer value: 1.8051818044922352%
        Average number of annoted chunks per file: 8.0
        Average number of chunks in use with correct footer and annotated per file:
            0.2927723263716544
        Set of sizes of key chunks: {32, 48, 64}
        Sizes of key chunks with their number of occurences:
        Size: 32  Number of occurences: 34366
        Size: 48  Number of occurences: 109346
        Size: 64  Number of occurences: 13434
        Number of sizes: 157146
        Number of unique sizes: {32, 48, 64}
```

Listing 2.23: Printing cleaned dataset chunk parsing global statistics.

The cleaned dataset contains 26191 RAW files and their corresponding annotation files. The program has been able to parse all those files, and has been able to detect 37682063 chunks, which represents 674232832 blocks. This is a huge number of blocks. The goal being to be able to predict which of those blocks are first key blocks, we need to be able to filter out the useless blocks as much as possible to both optimize computations and scale down the problem.

Using the `P` flag technique, we can see that 37346373 chunks are in use, and 354410 chunks are free. Although the proportion of free chunks is only 0.94%, there are 27.19% of the blocks that are in free chunks. More importantly, we can see that no free chunk is annotated. This means we can filter out all free chunks and their blocks. This allows a huge reduction of the scale of the problem.

The average number of annotated chunks per file being a perfect 8 value, this means that all the parsed files indeed contains the 6 key annotations with the additional `SSH_STRUCT` and `SSH_KEY` annotations. The dataset is very imbalanced since we have only 6 keys times the number of RAW files as positive labels and the rest as negative, thus the need for advanced reduction techniques.

The exact code to annotate the chunks can be as simple as the following:

Algorithm 11 Annotate Chunk Algorithm

procedure AnnotateChunk($chunk, keys\_addresses, ssh\_struct\_addr, session\_state\_addr$)
Ensure: $chunk$ object
Ensure: $keys\_addresses$ list of integers
Ensure: ssh_struct_addr integer
Ensure: $session\_state\_addr$ integer
    Note: Annotations should be done after free chunk detection.
    procedure AssertChunkUsedThenAnnotate($chunk, annotation$)
        Assert $chunk.is\_in\_use$ ▷ Make sure we don't annotate free chunks
        $chunk.annotations.append(annotation)$
    end procedure
    if $chunk.address \in keys\_addresses$ **then**
        AssertChunkUsedThenAnnotate($chunk, ChunkAnnotation.ChunkContainsKey$)
    else if $chunk.address = ssh\_struct\_addr$ **then**
        AssertChunkUsedThenAnnotate($chunk, ChunkAnnotation.ChunkContainsSSHStruct$)
    else if $chunk.address = session\_state\_addr$ **then**
        AssertChunkUsedThenAnnotate($chunk, ChunkAnnotation.ChunkContainsSessionState$)
    end if
end procedure

This algorithm in itself and the results observed is an important discovery. The annotations are actually always given for the `chunk.address` which corresponds to the address of the first block after the malloc header block. This means that the annotations are actually given for the beginning of the user data space of a chunk. This is crucial discovery, since it means that we can filter out the malloc header and footer blocks, and only keep the first block of the user data space of the chunks we want to embed. There are $674232832 - 183331224 = 490901608$ blocks in use. But there is only 37346373 chunks in use. This means that we can reduce the number of blocks to embed from 490901608 to 37346373 which is an additional reduction applied after the previous filtering that reduces the scale of the problem by a factor of 13.

Now let's look at the footer parsing. We can see in the logs that 94.72% of free chunks are said to have a correct footer value. But this value is misleading. Since the last chunk of a heap dump is often cropped, it means it has no footer. But we consider those special last chunks as free chunks. In fact, in the 354410 free chunks, we have 18720 or around 5.28% of them that are those special last cropped chunks only composed of zeros. With this perspective, we understand that 100% of the free chunks should be considered with correct footer value. In contrast, only 1.81% of in-use chunks have a correct footer value. It's tempting to think that maybe, those few chunks could maybe be actually empty too and removed. But this is not the case since a few chunks are actually both in-use, with a key annotation and a correct footer value. This means that we need to keep those chunks.

2.2.9.1 Chunk filtering

Based on the previous observations, we can propose different ways of filtering some chunks out. The objective is to reduce the number of chunks before any further processing to reduce the imbalanceness.

Since we have seen that free chunks are never annotated, we can filter them out. This filtering technique is applied in the final implementation of the chunk parsing algorithm. This filtering technique allows to reduce the number of chunks from 37682063 to 37346373, which is a reduction

of 0.89%. This is not a huge reduction, but it's still a reduction.

After all those extensive analysis and tests, we have gained invaluable knowledge about how we can reduce the scale of the problem and parse the files. Now, we need a way to create meaninglful embeddings for the blocks we want to perform machine learning on. This is the goal of the next section.

2.3 Graph-based memory dumps embedding

Now that we have a decent understanding of the dataset as well as the low level memory dump format, we can start to think about how to convert the memory dumps into graphs. As a recall, we want to be able to convert a memory dump into a graph representation that can be used for machine learning, since we want to be able to create a memory modelization as a basis for efficient embedding and feature engineering later. This is inherently due to the imbalanceness of the dataset, as we want to add more information to each memory block that just its raw bytes. The goal is to have a graph representation of the memory dump that can be used for efficient machine learning.

This section will be dedicated to describe the methods developped around this problem. We will first describe the design of the graph representation, then the implementation of the graph construction process. Finally, we will discuss the semantic embedding built from this graph representation.

2.3.1 Initial work from Python to Rust

Initially, we have been working and manipulating the code provided by SmartKex[3] for key detection. Our first explorations of the dataset quickly gave birth to some Jupyter Notebooks, which were used to explore the dataset and to understand the code, like `search_in_heap_mem.ipynb`. Rapidly, we decided to rebuild a complete Python 3.11 version of the code. This was done for several reasons:

- The provided code had no type hinting, which makes it hard to read and understand.
- We wanted to explore the dataset and learn by doing.
- The original code was not designed to be used as a library, but rather as a standalone script.
- The original code was just a few hundred lines of code and was not designed to be easily extensible, nor to be able to handle a large number of memory dumps.
- We wanted to modernize code by using the latest stable version of Python.

We decided to build a memory graph representation at that moment because we wanted to be able to add more information to the memory blocks than just their raw bytes. This new program was called `ssh_key_discover`, and relied on a number of Python libraries to work, like `graphviz`. This was a all-in-one library, composed of 2 sections, `mem_graph` and `ml_discovery`. The first one was devoted to build memory graphs, while the second one was dedicated to the data science and machine learning part.

[3] SmartKex GitHub repository: `https://github.com/smartvmi/Smart-and-Naive-SSH-Key-Extraction`

This initial program was already capable of handling several data processing pipelines, including machine learning pipelines with models like Random Forest, a grid search for hyperparameter optimization, and a cross validation pipeline, several balancing strategies and of course, a memory graph representation and semantic embedding. As an early development version, this program was not optimized for performance, and just loading a given heap dump file and its annotation, and then building the memory graph representation, could take from 30 seconds to a minute (on the TUXEDO machine), depending on the size of the heap dump file. As the original dataset comprises more than 10^6 files, a rapid estimation of the time needed just for the semantic embedding of the memory graph representation was above a month. In this regard, this initial program was just used on a bunch of files as a way to develop the semantic embedding model, algorithm and start working on feature engineering and machine learning. But it could not be used to produce final results on the whole dataset due to the performance issues described above.

This optimization issue was clearly not acceptable, and we decided to rewrite the graph part in Rust. This is a compiled language that leverages zero-cost abstractions, and thus, is several order of magnitude faster than Python. This was also a good opportunity to learn Rust, which is a language that is gaining more and more popularity, especially in the security community. This new program was called `mem2graph`. Switching from Rust to Python and doing a proper use of multithreading allowed us to reduce the time needed to build the memory graph representation from 30 seconds to less than 1 second. In out case, and comparing using only the TUXEDO laptop, this represent an estimated minimum of a 130x speedup. But this is even much better on the server, where the multithreading can really be leverage. This was a significant improvement which allowed us to build the memory graph representation for the whole original dataset in a just a few hours.

2.3.2 Memory Graph Representation

Now, let's describe the memory graph representation. The goal is to be able to represent a memory dump as a graph. This modelization makes sense since the heap dump can be considered as having memory chunks as nodes, being connected by pointers acting as arrows. This is a very natural way to represent a memory dump. However in our cases, and since the goal is to make predictions on raw bytes, we will not use the chunks as nodes, but rather the memory blocks directly. This is because we want to be able to make predictions on raw blocks of bytes, and not on chunks.

Our memory graph representation is composed of a directed graph, where each node is a memory block of bytes, and each edge is either indicative of a pointer link or a chunk membership relationship. This second representation is directly inspired by collection representation in Knownledge Graph ontologies. In the case of RDF, this could be equivalent to a **rdf:Bag**, which is an unordered container [5] (see **??**). The graph is directed because the pointers are directed. We will also consider the relationship of belonging to a chunk as oriented from the data structure header block to the data structure member blocks.

Our memory graph representation is inherently a property graph. Each node and edge can have properties. The properties of an edge are the type of the edge, which can be either a pointer or a structure membership relationship.

- **dts:** Data Structure Membership Relationship
- **ptr:** Pointer Relationship (direction is from the source to the target)

In our case, the properties of a node are at minimum the address and the byte block. The graph is also heterogeneous since our nodes can have different types corresponding to their infered characteristics.

- **PN:** Pointer Node. This is a node whose bytes have been identified as a pointer.
- **DTN:** Data Structure Node. This is a node whose bytes have been identified as a data structure header. In the graph, this node is the root node of an allocated structure.
- **KN:** Key Node. This is a node whose bytes have been identified as a key. This identification relies both on the annotations and some verification checks.
- **VN:** Value Node. These are all blocks that have not been identified. It is the default node type.

These nodes and edges form the base of the memory graph representation. Below is a simplified (truncated) example of a memory graph representation. The full example is available in .2. For clarity, the addresses are not displayed in this simplified version. Another version of this graph with real addresses is available in .2.

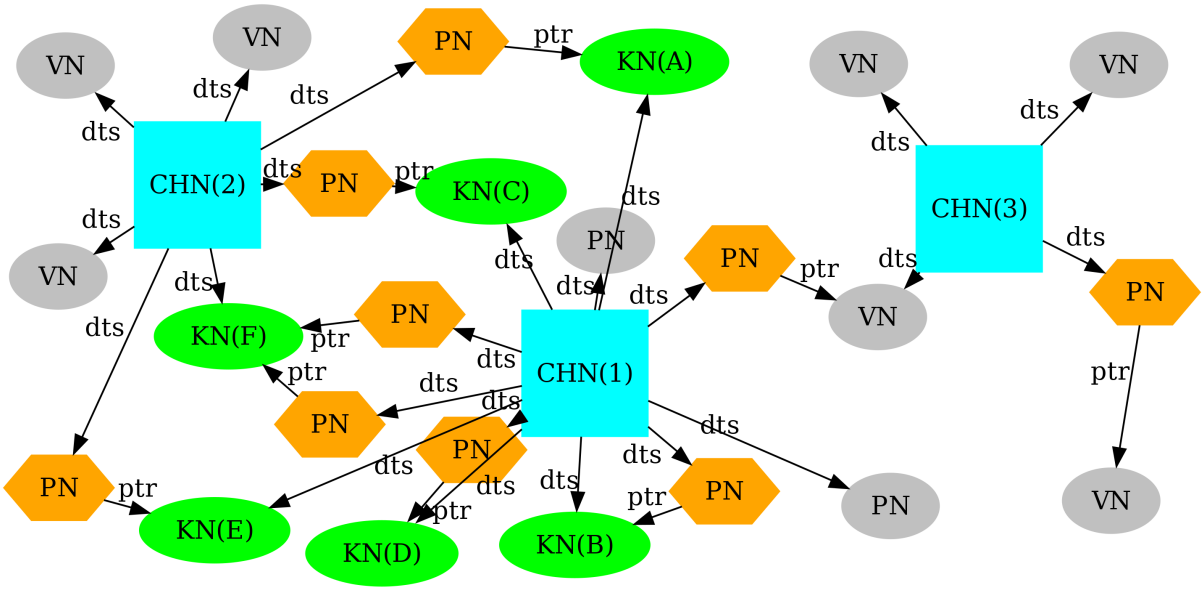


Figure 2.7: Visualization of a truncated memory graph generated from *Training/basic/V_7_1_-P1/24/17016-1643962152-heap.raw*. The addresses are not displayed for imploved readability. Version with addresses here .2.

The given graph represents a memory layout with various types of nodes, each serving a specific purpose. The graph contains CHN nodes, which act as the root nodes for allocated structures and are colored in cyan. These DTN nodes are connected to KN nodes, which are identified as keys and are colored in green. The PN nodes, colored in orange, are pointers and can be connected to value nodes or key nodes. Finally, the graph includes VN nodes, which are the default node types and are colored in grey. These nodes have not been identified as any specific type and may contain arbitrary values.

The idea behind this representation will be to try to make predictions on the KN nodes, which are the nodes that have been identified as keys. Using the graph, we can build an embedding of

the nodes and as such, add more information to a given byte block than just its raw bytes. This is the basis of the semantic embedding, which will be discussed later.

This example is based on the heap dump file *Training/basic/V_7_1_P1/24/17016-1643962152-heap.raw* and has been generated using `mem2graph`, and the **sfdp** layout algorithm from `graphviz` using the following command:

```
sfdp -Gsize=30! -Goverlap=voronoi -Tpng 17016-1643962152_truncated_no_addresses.gv >
    17016-1643962152_truncated_no_addresses.png
```

Listing 2.24: Command used to generate the memory graph visualization of *Training/basic/V_7_1_P1/24/17016-1643962152-heap.raw*

2.4 From heap dump to memory graph embeddings

Now that the basis of the memory graph representation has been described, let's dive in the different phases involved in transforming a raw heap dump file into a memory graph file with some custom embeddings that can later be loaded and used by some data analysis and ML programs.

2.4.0.1 Initialisation and data checking

1. graph initialization * loading a given heap dump file and its associated annotation file * perform some checks on the annotation file, like checking if the annotation file is valid, that all annotations are present

2.4.0.2 Graph Construction steps

2. graph building * build the graph from heap dump byte blocks * perform data structure detection step * perform pointer detection step

Optionally perform chunk pointer reduction step, to remove any blocks that are not Chunk Header Nodes, that in the context of the memory graph, represent the root node of a chunk.

2.4.0.3 Graph annotation

3. graph annotation replace VN by KN from annotation file add other annotations like SSH_STRUCT After this step, its possible to export the graph to a file, like a `.dot` file or a `.gv` file for visualization or other purposes.

2.4.0.4 Custom graph-based embeddings

4. Generate embedding from graph different embedding are possible. We will focus on the semantic embedding which is a general way to embed the graph by addind the related graph structure and vicinity information to each node.

2.4.0.5 Exporting the Graph

5. Exporting the graph export the graph to a .gv DOT file, with custom embeddings being integrated by using the comment fields and a slightly modified JSON format. This allows to easily read the embeddings associated with each node, and is still valid DOT file that can be used through tool and libraries supporting DOT graph formal.

Explain what is the dot format, and show examples.

2.5 A wilde range of features and embeddings

2.5.1 Embeddings based on custom features

2.5.1.1 Semantic Embedding

2.5.1.2 Statistical Embedding

2.5.1.3 Start-bytes Embedding

2.5.1.4 Node filtering to feature

2.5.2 Graph-agnostic embeddings

2.5.2.1 RandomWalk

2.5.2.2 Node2Vec

2.5.3 Feature evaluation

2.6 Machine learning binary classification

2.6.1 Classic models of machine learning

2.6.1.1 Random Forest

2.6.2 GCN

2.6.3 A first GCN model

2.6.4 The impact of complexity

2.6.5 More advanced models

2.7 ML for key prediction

The chapter has been an overview of the dataset, development environment, and tools used for this thesis. In the next chapter, we will dive deeper into the graph memory representation and associated algorithms and programs.

3 Results

The following section describes the experimental setup, the used datasets and parameters and the experimental results achieved.

3.1 Developped programs

3.1.1 Mem2Graph

3.1.2 Machine Learning pipelines

3.2 Experimental Setup

3.2.1 Working with a huge dataset

3.2.2 Dealing with hyperparameter tuning

3.2.3 The challenge of ressource optimisation

3.2.4 The challenge of packet management

3.3 Obtained results on feature engineering

```
1  (py311) root@compute-container-rascoussie-d584d4794-lbm9r:~/onyr_phdtrack/mem2graph#
       find data/ -type f -name "*.gv" | wc -l
2  104808
```

Listing 3.1: Command used to count the number of .gv memory graph files generated by *mem2graph* inside one of the servers mem2graph dataset directory.

3.4 Feature Engineering results

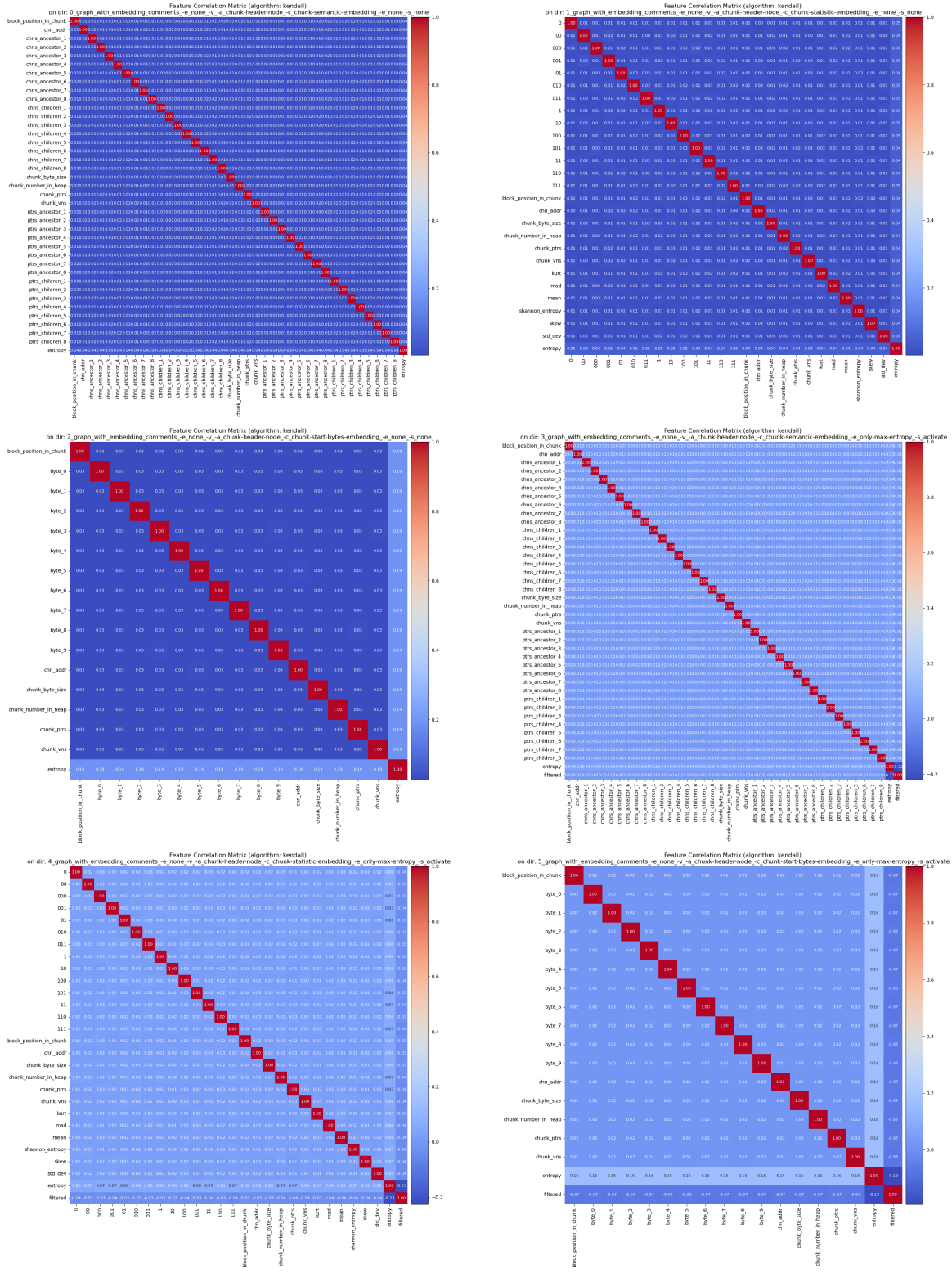


Figure 3.1: Feature correlation matrices on the different Mem2Graph-generated datasets. Used algorithm: Kendall.

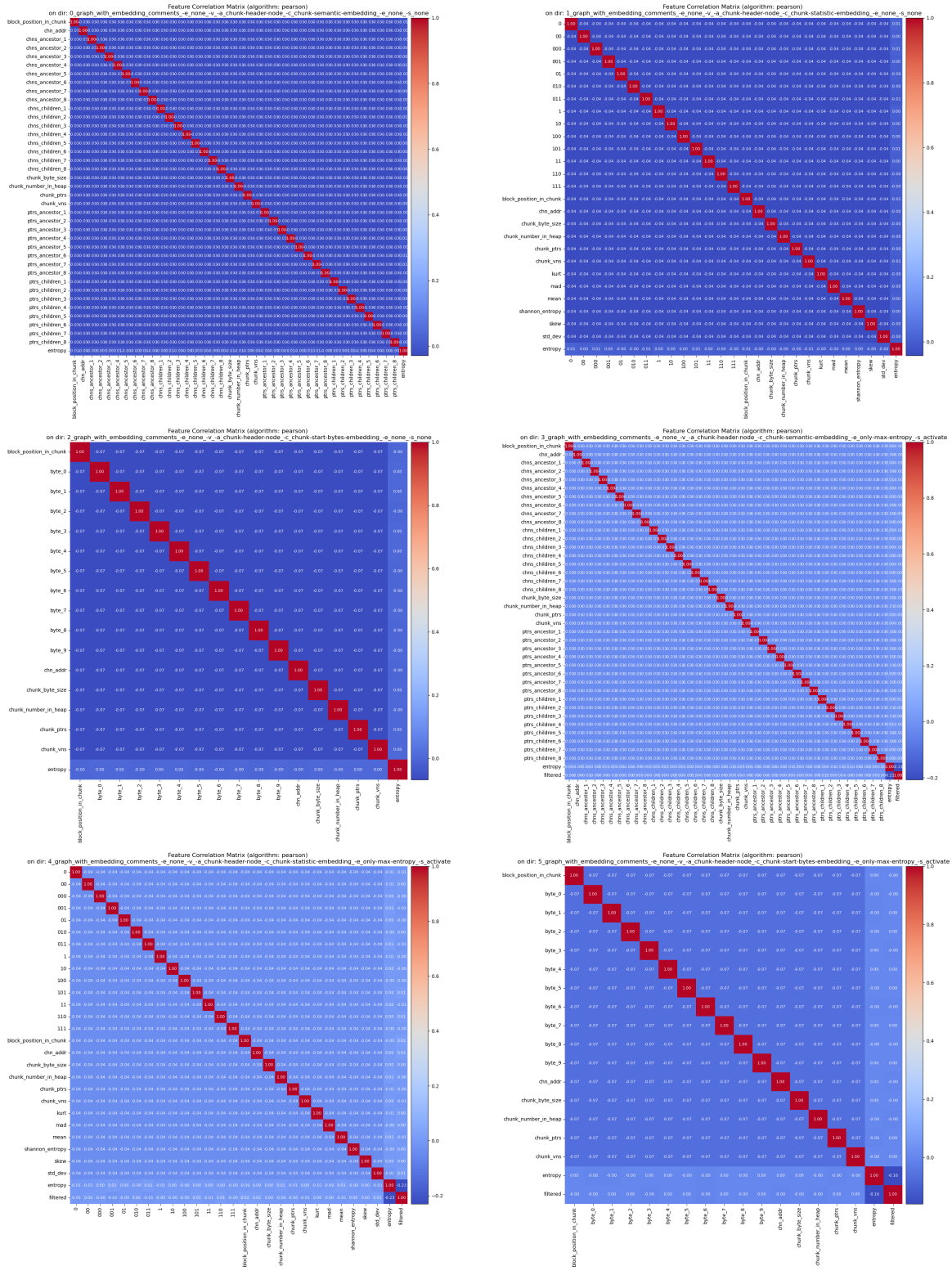


Figure 3.2: Feature correlation matrices on the different Mem2Graph-generated datasets. Used algorithm: Pearson.

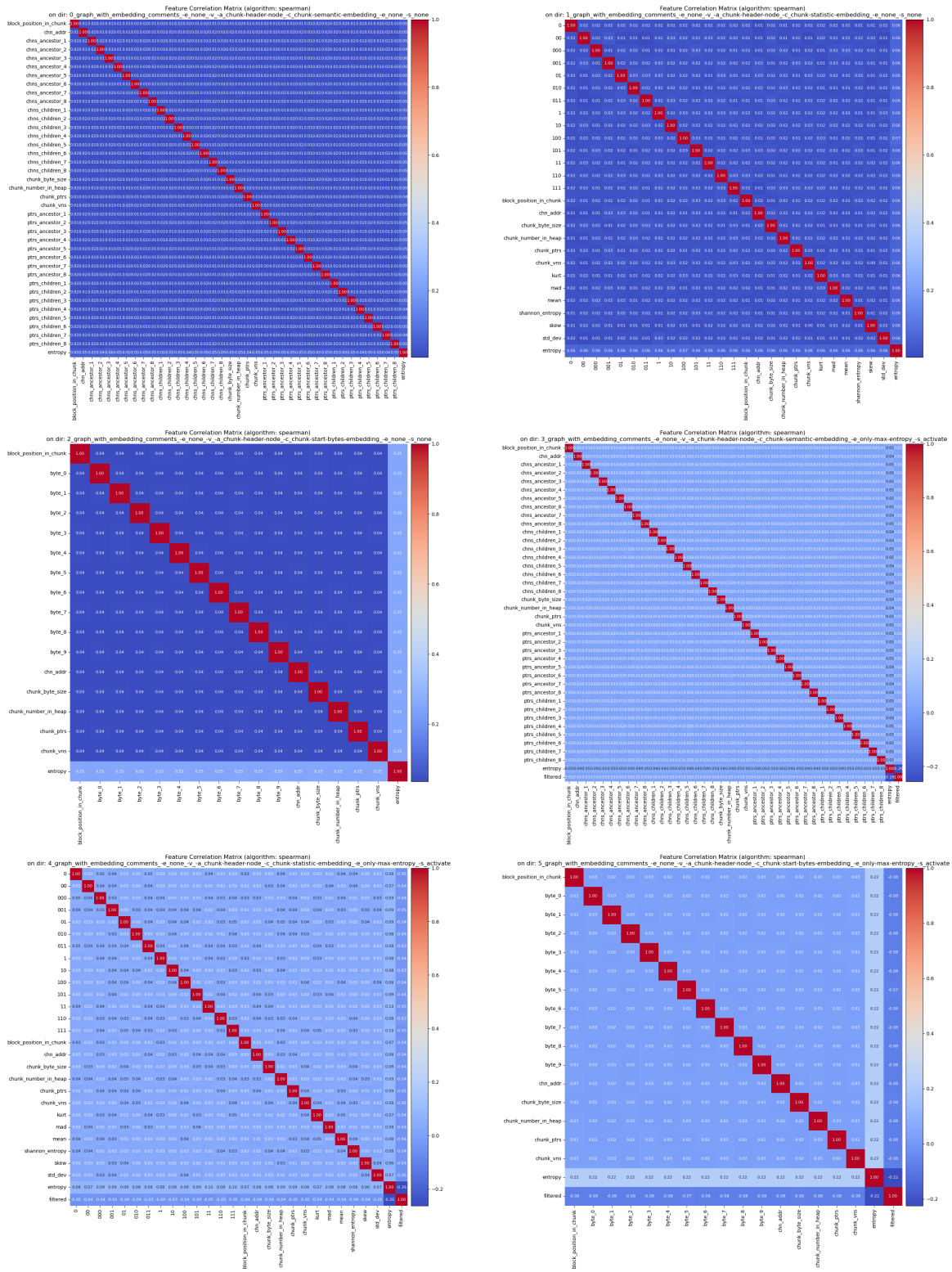


Figure 3.3: Feature correlation matrices on the different Mem2Graph-generated datasets. Used algorithm: Spearman.

3.5 Classic Model results

3.6 Deep Learning Model results

4 Discussion

Discuss the results. What is the outcome of your experiments?

4.1 Objectives of the experiments

4.2 Discussing features and embeddings

4.3 Comparing GCN and classical ML models

5 Conclusion

The evolving landscape of cybersecurity necessitates robust techniques for safeguarding digital communications. OpenSSH, a pivotal element in this landscape, is a popular implementation of the Secure Shell (SSH) protocol, which enables secure communication between two networked devices. The protocol is widely used in the industry, particularly in the context of remote access to servers. Using digital forensic techniques, it is possible to extract the SSH keys from memory dumps, which can then be used to decode encrypted communications thus allowing the monitoring of controlled systems. At the crux of this Masterarbeit is the development of algorithms and machine learning models to predict SSH keys within these heap dumps, focusing on using graph-like-structures and vectorization for custom embeddings. With an interdisciplinary approach that fuses traditional feature engineering with graph-based methods as well as memory modelization for inductive reasoning and learning inspired by recent developments in Knownledge Graph (KG)s, this research not only leverages existing machine learning paradigms but also explores new avenues, such as Graph Convolutional Networks (GCN) applied to memory forensics. The present work also introduces a new memory forensics tool, *mem2graph*, which is designed to be modular and extensible, and which can be used to generate memory graphs from memory dumps.

5.1 Summary of Results

Below is a summary of the results achieved in the present work.

5.1.1 Dataset Exploration

A careful exploration of the dataset, and deep understanding of the original heap dumps have been invaluable in discovering patterns in the raw data. This exploration has allowed the development of a range of parsing algorithm able to extract information like structure and content of a given heap dump.

It has been discovered that the problem of finding the address of keys in the heap dump can be reduced to identifying the chunks that contain those keys. This allows to reduce the size of the problems from aroud 100 000 of blocks per heap dump, to around 1000 chunks per file. This also allows to concentrate the heap dump memory graph representation around the chunks.

It has also been demonstrated that two powerful chunk filtering techniques can drastically reduce the number of chunks to consider. The first filter criterion consist in the Shannon's entropy value of a chunk user data start bytes. This is because the keys are expected to have a high entropy compared to other raw data types. The second important criterion is the chunk byte size. It has been shown that key chunks actually have a small size in the range of possible key size. If filtering is not possible, as it is the case with GCN models, those filters can actually be converted in powerful float and boolean features.

5.1.2 Memory Graph Generation

This masterarbeit has introduced a range of algorithms able to generate memory graphs from memory dumps. The algorithms are designed to be as generic as possible, and can be applied to

any memory dump dataset. The algorithms are mostly implemented in the *mem2graph* program, and many exploration and sanity checking scripts are also available in Python.

With those algorithms, it is possible to parse a RAW heap dump file, and transform it into a memory graph. The memory graph is a graph-like structure, where each node represents a memory block with a precise address in the heap. Each edge represents either a pointer pointing to another block, or materialize the fact that a block belongs to a specific chunk. In order to reduce the size of the graph, it is possible to compact the block graph into a chunk graph, where each node represents a chunk, and chunks are connected through their pointers.

5.1.3 Feature Engineering and Embeddings

The memory graph can be used to extract features from the memory dump, and to apply machine learning algorithms to the memory dump. It can also be used for direct graph visualization. The memory graph serves as a direct source of embedding whether they are made manually or using readily available and tested techniques like RandomWalks or Node2Vec.

All those embeddings can be combined. The feature evaluation has shown that those features are very lowly correlated, meaning that their quality is high. However, all those different embeddings doens't have the same results on the ML and GCN models, depending on the strengths and weaknesses of the different models.

5.1.4 Classic Machine Learning Models

5.1.5 Deep Learning GCN Models

5.2 Outlook on Future Work

The current report, in conjunction with the associated Masterarbeit, has introduced numerous novel algorithms and implementations. These have been instrumental in addressing the initial research questions. However, as with most research endeavors, new queries and potential avenues for enhancement have emerged, paving the way towards further exploration.

The methodologies and algorithms introduced for the OpenSSH memory dump dataset are versatile and can be extended to other memory dump datasets utilizing the GLIBC library. Given that this library is the default for Linux, adapting the methods from this Masterarbeit to other applications requires minimal effort. The *mem2graph* program is inherently modular and built for extensibility. Furthermore, this tool can be employed to produce memory graphs for diverse datasets. Thanks to the universal character of the generated embeddings and memory graphs, new datasets can be readily integrated into the ML and DL pipelines crafted in Python. While an extensive array of features and embedding techniques have been explored in this report, there remains ample opportunity for innovative experimentation.

For a seamless fusion of machine learning into the *mem2graph* program, further effort is required. Embedding machine learning immediately post-memory graph creation can substantially boost efficiency, particularly when aiming to craft a real-time OpenSSH memory forensics utility. However, this integration is challenging due to the current limited ML support within Rust.

Another avenue for enhancement involves analyzing the effects of different C libraries on

allocated chunks and the layout of heap dump memory. Investigating various languages could also be insightful. Depending on the level of variation encountered, modifications to the algorithms might be required, especially concerning the architecture involved in generating or extracting heap dump configurations. Pursuing this direction could significantly advance the development of a universal machine learning-assisted memory forensics tool for key extraction.

While the background section underscores the vast array of ML architectures available, it's clear that not all can be thoroughly explored. This research has primarily addressed the most common and promising ones, yet numerous others await investigation. The tools crafted to bolster ML pipelines present a solid foundation for such endeavors. Another dimension to consider is hyperparameter optimization. Given the constraints of time and resources, only certain parameter ranges were tested. Expanding these tests, incorporating larger datasets, and harnessing increased computational capacity can directly enhance performance.

.1 Code

.2 Memory Graphs

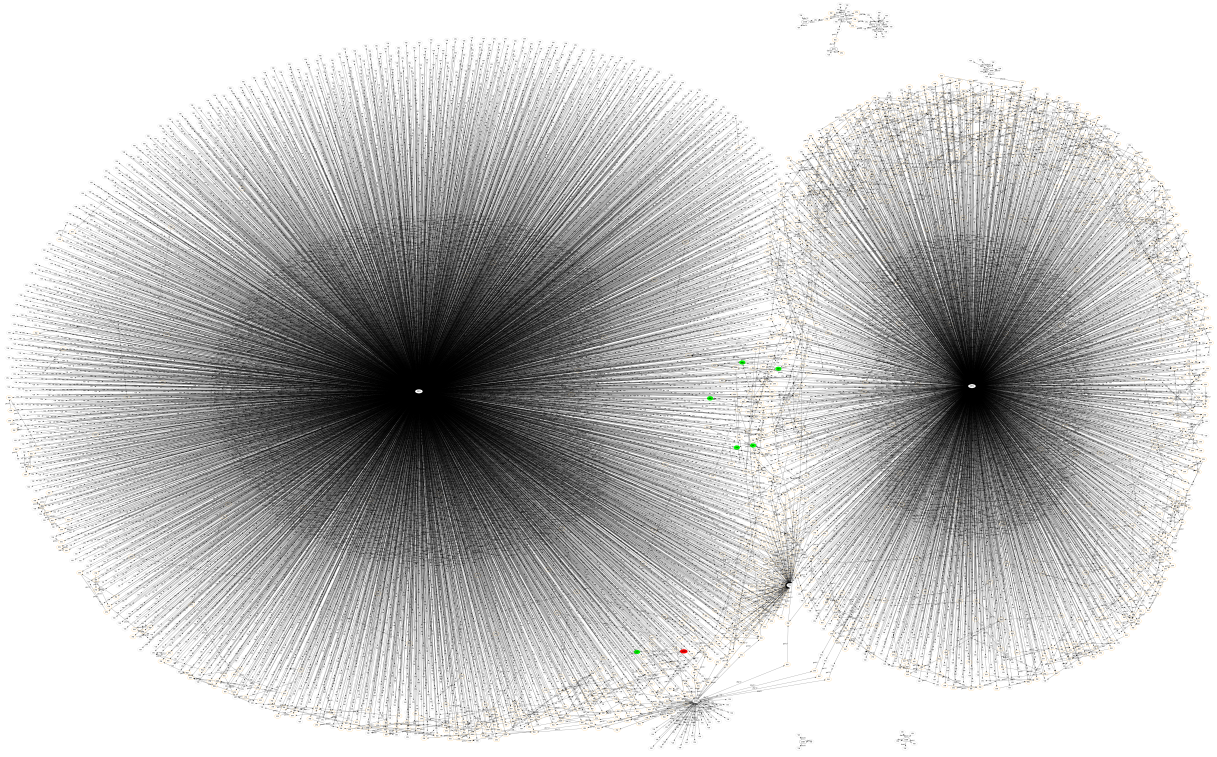


Figure 1: Visualization of the full memory graph generated from *Training/basic/V_7_1_-P1/24/17016-1643962152-heap.raw*.

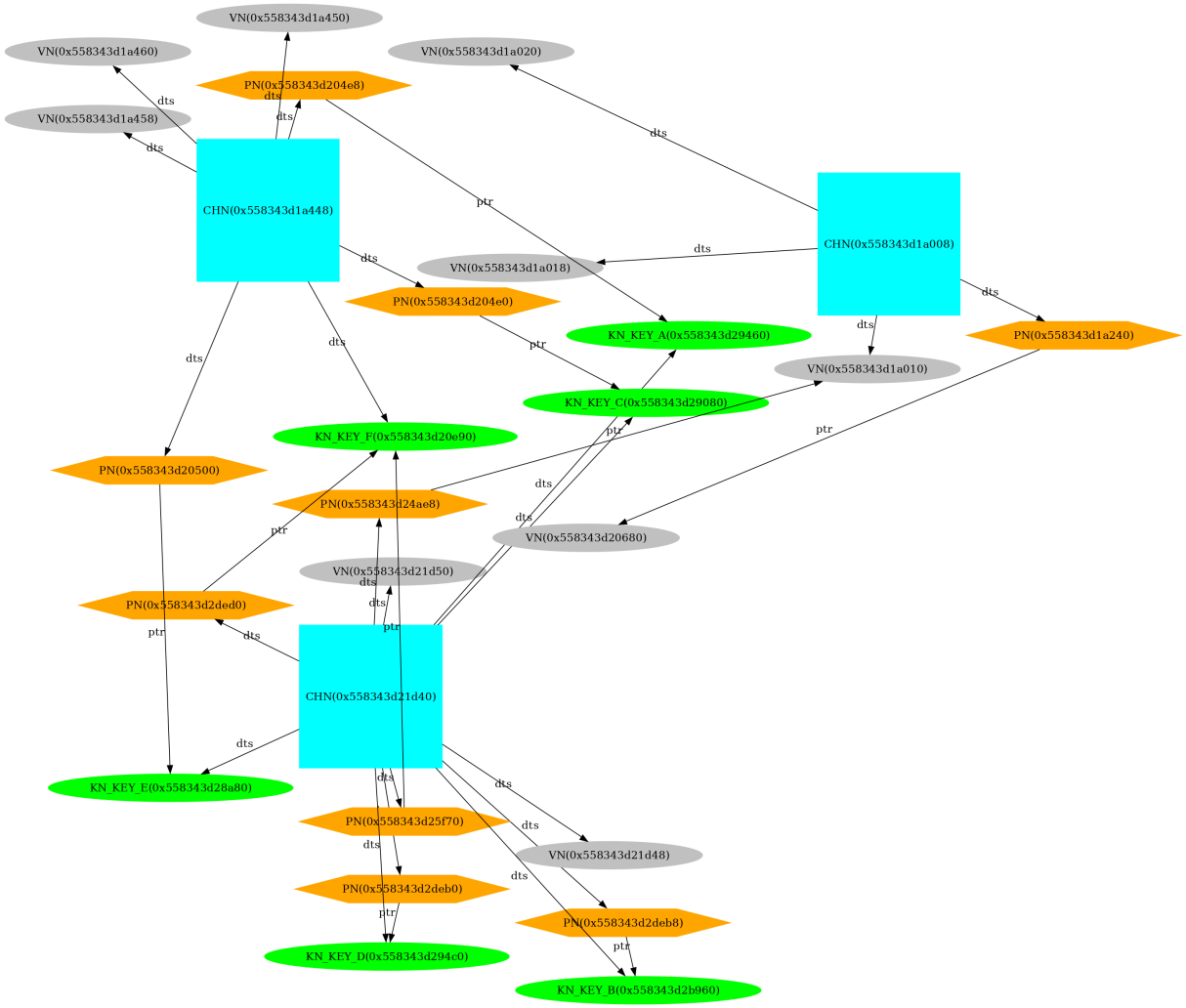


Figure 2: Visualization of a truncated memory graph generated from *Training/basic/V_7_1_-P1/24/17016-1643962152-heap.raw*. Here with real addresses.

Generated using a slightly different command, for better layout of the nodes:

```
sfdp -Gsize=30! -Goverlap=ortho -Tpng 17016-1643962152_truncated.gv >
    17016-1643962152_truncated.png
```

Listing 1: Command used to generate the memory graph visualization of *Training/basic/V_7_1_-P1/24/17016-1643962152-heap.raw* here using real addresses.

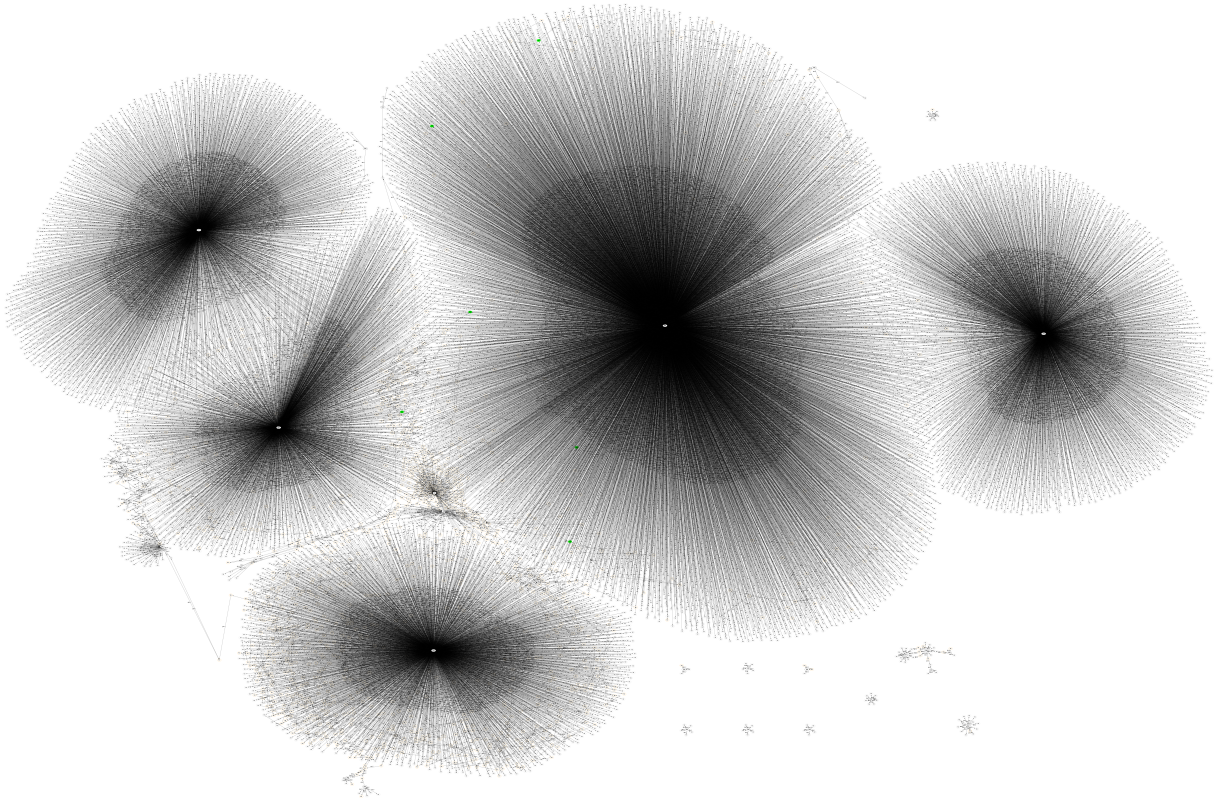


Figure 3: Visualization of the full memory graph generated from *Training/scp/V_7_8_P1/16/302-1644391327-heap.raw*.

.3 Dataset

Acronyms

AI Artificial Intelligence. 18

DEL Directed Edge-labelled Graphs. 16

ESS Estimated Security Strength. 8

GCN Graph Convolutional Networks. 2, 67

GNN Graph Neural Network. 32, 33

GRU Gated Recurrent Units. 31

KG Knownledge Graph. 12, 16, 17, 62, 67

KNN K-Nearest Neighbors. 20

LDA Linear Discriminant Analysis. 20

LLM Large Language Model. 24

LSB Least Significant Bit. 57

ML Machine Learning. 2, 18, 20, 28, 29, 45, 54–56, 58–60, 67

NLP Natural Language Processing. 22

OWL Web Ontology Language. 17

PCA Principal Component Analysis. 20

RDF Resource Description Framework. 16, 17, 62

RNN Recurrent Neural Networks. 30

SMOTE Synthetic Minority Over-sampling Technique. 37

SSH Secure Shell Protocol. ii, 5, 8, 67

SVM Support Vector Machine. 20

t-SNE t-distributed Stochastic Neighbor Embedding. 20

VMI Virtual Machine Introspection. 36

Glossary

. 63

KN Key Node. This is a node whose bytes have been identified as a key. This identification relies both on the annotations and some verification checks.. 63, 64

PN Pointer Node. This is a node whose bytes have been identified as a pointer.. 63

VN Value Node. These are all blocks that have not been identified. It is the default node type.. 63

References

[1] Stewart Sentanoe and Hans P. Reiser. „SSHkex: Leveraging virtual machine introspection for extracting SSH keys and decrypting SSH network traffic“. en. In: *Forensic Science International: Digital Investigation* 40 (2022), p. 301337. doi: 10.1016/j.fsidi.2022.301337. url: https://linkinghub.elsevier.com/retrieve/pii/S2666281722000063.

[2] Wolfram Gloger and Doug Lea. *Malloc implementation for multiple threads without lock contention*. Accessed: 2023-09-22. Free Software Foundation, Inc. 2001. url: https://elixir.bootlin.com/glibc/glibc-2.28/source/malloc/malloc.c (visited on 09/22/2023).

[3] DJ Delorie et al. *Malloc Internals*. Accessed: 2023-09-22. Sourceware. 2023. url: https://sourceware.org/glibc/wiki/MallocInternals (visited on 09/25/2023).

[4] Unknown. *How does glibc malloc work?* Asked 6 years, 6 months ago; Modified 3 years, 6 months ago; Viewed 11k times; Accessed: 2023-09-25. 2023. url: https://reverseengineering.stackexchange.com/questions/15033/how-does-glibc-malloc-work/15038#15038.

[5] Joep Meindertma. *Ordered data in RDF: About Arrays, Lists, Collections, Sequences and Pagination*. Accessed: 2023-09-22. Feb. 7, 2020. url: https://ontola.io/blog/ordered-data-in-rdf (visited on 09/22/2023).

[6] Christofer Fellicious et al. „SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump“. In: arXiv:2209.05243 (Sept. 2022). arXiv:2209.05243 [cs]. doi: 10.48550/arXiv.2209.05243. url: http://arxiv.org/abs/2209.05243.

[7] Eelco Dolstra. *The purely functional software deployment model*. Utrecht University, 2006. url: https://edolstra.github.io/pubs/phd-thesis.pdf.

[8] OpenScience ASAP. *Was ist Open Science?* Accessed: 2023/09/19. 2023. url: http://openscienceasap.org/open-science/ (visited on 09/19/2023).

[9] DGE Gomes et al. „Why don’t we share data and code? Perceived barriers and benefits to public archiving practices“. In: *Proc Biol Sci* 289.1987 (Nov. 30, 2022), p. 20221113. doi: 10.1098/rspb.2022.1113. eprint: Epub2022Nov23.

[10] Eelco Dolstra and Andres Löh. „NixOS: A Purely Functional Linux Distribution“. In: *SIGPLAN Not.* 43.9 (Sept. 2008), pp. 367–378. issn: 0362-1340. doi: 10.1145/1411203.1411255. url: https://doi.org/10.1145/1411203.1411255.

[11] José Tomás Martínez Garre, Manuel Gil Pérez, and Antonio Ruiz-Martínez. „A novel Machine Learning-based approach for the detection of SSH botnet infection“. In: *Future Generation Computer Systems* 115 (Feb. 2021), pp. 387–396. doi: 10.1016/j.future.2020.09.004. url: https://www.sciencedirect.com/science/article/pii/S0167739X20303265.

[12] Sanjay Madan and Monika Singh. „Classification of IOT-Malware using Machine Learning“. In: *2021 International Conference on Technological Advancements and Innovations (ICTAI)*. Nov. 2021, pp. 599–605. doi: 10.1109/ICTAI53825.2021.9673185.

[13] Connor Hetzler, Zachary Chen, and Tahir M. Khan. „Analysis of SSH Honeypot Effectiveness“. en. In: *Advances in Information and Communication*. Ed. by Kohei Arai. Lecture Notes in Networks and Systems. Cham: Springer Nature Switzerland, Mar. 2023, pp. 759–782. isbn: 9783031280733. doi: 10.1007/978-3-031-28073-3_51.

Additional bibliography

[14] M. Baushke. *Key Exchange (KEX) Method Updates and Recommendations for Secure Shell (SSH)*. RFC 9142. Updates: 4250, 4253, 4432, 4462; Errata exist. Internet Engineering Task Force (IETF), Jan. 2022.

[15] Nicole Perlroth, Jeff Larson, and Scott Shane. „N.S.A. Able to Foil Basic Safeguards of Privacy on Web“. In: *The New York Times* (Sept. 2013). url: https://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html.

[16] James Ball, Julian Borger, and Glenn Greenwald. „Revealed: how US and UK spy agencies defeat internet privacy and security“. In: *The Guardian* (Sept. 2013). Published at 11.24 BST. url: https://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security.

[17] Aris Adamantiadis. *OpenSSH introduces curve25519-sha256@libssh.org key exchange !* Retrieved 2023-09-05. Nov. 2013. url: https://www.libssh.org/2013/11/03/openssh-introduces-curve25519-sha256libssh-org-key-exchange/.

[18] OpenSSH. *OpenSSH 5.7 release notes*. Retrieved 2022-11-13. Jan. 2011. url: https://www.openssh.com/txt/release-5.7.

[19] Stewart Sentanoe, Benjamin Taubmann, and Hans P. Reiser. „Sarracenia: Enhancing the Performance and Stealthiness of SSH Honeypots Using Virtual Machine Introspection“. In: *Lecture Notes in Computer Science*. Vol. 11252. Conference paper. Nov. 2018. url: https://link.springer.com/chapter/10.1007/978-3-030-03638-6_16.

[20] OpenSSH. *OpenSSH 6.5 release notes*. Retrieved 2022-11-13. Jan. 2014. url: https://www.openssh.com/txt/release-6.5.

[21] OpenSSH. *OpenSSH 7.0 release notes*. Retrieved 2022-11-13. Aug. 2015. url: https://www.openssh.com/txt/release-7.0.

[22] Christine Solnon. „Théorie des graphes et optimisation dans les graphes“. In: *INSA de Lyon* ().

[23] Douglas Brent West et al. *Introduction to graph theory*. Vol. 2. Prentice hall Upper Saddle River, 2001.

[24] OpenSSH. *OpenSSH 7.2 release notes*. Retrieved 2022-11-13. Jan. 2016. url: https://www.openssh.com/txt/release-7.2.

[25] OpenSSH. *OpenSSH 8.2 release notes*. Retrieved 2022-11-13. Feb. 2020. url: https://www.openssh.com/txt/release-8.2.

[26] OpenSSH. *OpenSSH 8.8 release notes*. Retrieved 2022-11-13. Sept. 2021. url: https://www.openssh.com/txt/release-8.2.

[27] Samina Khalid, Tehmina Khalil, and Shamila Nasreen. „A survey of feature selection and feature extraction techniques in machine learning“. In: *2014 Science and Information Conference*. Aug. 2014, pp. 372–378. doi: 10.1109/SAI.2014.6918213.

[28] Sinan Ozdemir and Divya Susarla. *Feature Engineering Made Easy: Identify unique features from your dataset in order to build powerful machine learning systems*. Packt Publishing Ltd, 2018.

[29] C E Shannon. „A Mathematical Theory of Communication“. en. In: *The Bell System Technical Journal* 27 (Oct. 1948), pp. 379–423.

[30] F. Pedregosa et al. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[31] Richard Boddy and Gordon Smith. *Statistical methods in practice: for scientists and technologists*. John Wiley & Sons, 2009.

[32] Michael I Jordan and Tom M Mitchell. „Machine learning: Trends, perspectives, and prospects“. In: *Science* 349.6245 (2015), pp. 255–260. url: https://www.science.org/doi/full/10.1126/science.aaa8415.

[33] Todd G Nick and Kathleen M Campbell. „Logistic regression“. In: *Topics in biostatistics* (2007). Publisher: Springer, pp. 273–301.

[34] S. B. Kotsiantis. „Decision trees: a recent overview“. In: *Artificial Intelligence Review* 39.4 (Apr. 2013), pp. 261–283. issn: 0269-2821, 1573-7462. doi: 10.1007/s10462-011-9272-4. url: http://link.springer.com/10.1007/s10462-011-9272-4 (visited on 08/30/2023).

[35] Philipp Probst, Marvin Wright, and Anne-Laure Boulesteix. „Hyperparameters and Tuning Strategies for Random Forest“. In: *WIREs Data Mining and Knowledge Discovery* 9.3 (May 2019), e1301. issn: 1942-4787, 1942-4795. doi: 10.1002/widm.1301. arXiv: 1804.03515[cs,stat]. url: http://arxiv.org/abs/1804.03515 (visited on 08/30/2023).

[36] Yann LeCun et al. „Gradient-Based Learning Applied to Document Recognition“. In: *proc of the IEEE* (1998).

[37] Dai Quoc Nguyen et al. „A Novel Embedding Model for Knowledge Base Completion Based on Convolutional Neural Network“. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Association for Computational Linguistics, 2018. doi: 10.18653/v1/n18-2053. url: https://doi.org/10.18653%2Fv1%2Fn18-2053.

[38] Daixin Wang, Peng Cui, and Wenwu Zhu. „Structural Deep Network Embedding“. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1225–1234. isbn: 9781450342322. doi: 10.1145/2939672.2939753. url: https://doi.org/10.1145/2939672.2939753.

[39] Michael Schlichtkrull et al. „Modeling relational data with graph convolutional networks“. In: *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*. Springer. 2018, pp. 593–607. url: https://arxiv.org/pdf/1703.06103.pdf.

[40] Liang Yao, Chengsheng Mao, and Yuan Luo. „KG-BERT: BERT for knowledge graph completion“. In: *arXiv preprint arXiv:1909.03193* (2019). url: https://arxiv.org/pdf/1909.03193.pdf.

[41] Ye Liu et al. „Kg-bart: Knowledge graph-augmented bart for generative commonsense reasoning“. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 7. 2021, pp. 6418–6425. url: file:///home/onyr/Downloads/16796-Article%20Text-20290-1-2-20210518.pdf.

[42] Aditya Grover and Jure Leskovec. „node2vec: Scalable feature learning for networks“. In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016, pp. 855–864. url: https://dl.acm.org/doi/pdf/10.1145/2939672.2939754.

[43] Jian Tang et al. „Line: Large-scale information network embedding“. In: *Proceedings of the 24th international conference on world wide web*. 2015, pp. 1067–1077. url: https://dl.acm.org/doi/pdf/10.1145/2736277.2741093.

[44] Petar Velickovic et al. „Graph attention networks“. In: *stat* 1050.20 (2017), pp. 10–48550. url: https://personal.utdallas.edu/~fxc190007/courses/20S-7301/GAT-questions.pdf.

[45] Keiron O'Shea and Ryan Nash. „An Introduction to Convolutional Neural Networks". In: *CoRR* abs/1511.08458 (2015). arXiv: 1511.08458. url: http://arxiv.org/abs/1511.08458.

[46] Jamie Ludwig. „Image convolution". In: *Portland State University* (2013). url: https://web.pdx.edu/~jduh/courses/Archive/geog481w07/Students/Ludwig_ImageConvolution.pdf.

[47] Vic Degraeve et al. „R-GCN: the R could stand for random". In: *arXiv:2203.02424 preprint* (2022). url: https://arxiv.org/pdf/2203.02424.pdf.

[48] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. „Deepwalk: Online learning of social representations". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2014, pp. 701–710. url: https://dl.acm.org/doi/pdf/10.1145/2623330.2623732.

[49] Will Hamilton, Zhitao Ying, and Jure Leskovec. „Inductive representation learning on large graphs". In: *Advances in neural information processing systems* 30 (2017). Ed. by I. Guyon et al. url: https://proceedings.neurips.cc/paper_files/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf.

[50] Zonghan Wu et al. „A comprehensive survey on graph neural networks". In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24. url: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9046288.

[51] Zonghan Wu et al. „Beyond low-pass filtering: Graph convolutional networks with automatic filtering". In: *IEEE Transactions on Knowledge and Data Engineering* (2022). url: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9806316.

[52] Federico Monti et al. „Geometric deep learning on graphs and manifolds using mixture model cnns". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 5115–5124. url: https://openaccess.thecvf.com/content_cvpr_2017/papers/Monti_Geometric_Deep_Learning_CVPR_2017_paper.pdf.

[53] Franco Scarselli et al. „The graph neural network model". In: *IEEE transactions on neural networks* 20.1 (2008), pp. 61–80. url: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4700287.

[54] Joan Bruna et al. „Spectral networks and locally connected networks on graphs". In: *arXiv preprint arXiv:1312.6203* (2013). url: https://arxiv.org/pdf/1312.6203.pdf.

[55] Quoc V Le et al. „A tutorial on deep learning part 2: Autoencoders, convolutional neural networks and recurrent neural networks". In: *Google Brain* 20 (2015), pp. 1–20. url: https://ai.stanford.edu/~quocle/tutorial2.pdf.

[56] Thomas N Kipf and Max Welling. „Semi-supervised classification with graph convolutional networks". In: *arXiv preprint arXiv:1609.02907* (2016). url: https://arxiv.org/pdf/1609.02907.pdf.

[57] Junyoung Chung et al. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. Dec. 11, 2014. arXiv: 1412.3555[cs]. url: http://arxiv.org/abs/1412.3555 (visited on 08/23/2023).

[58] J. Laaksonen and E. Oja. „Classification with learning k-nearest neighbors". In: *Proceedings of International Conference on Neural Networks (ICNN'96)*. International Conference on Neural Networks (ICNN'96). Vol. 3. Washington, DC, USA: IEEE, 1996, pp. 1480–1483. isbn: 978-0-7803-3210-2. doi: 10.1109/ICNN.1996.549118. url: http://ieeexplore.ieee.org/document/549118/ (visited on 08/30/2023).

[59] Sepp Hochreiter and Jürgen Schmidhuber. „Long short-term memory". In: *Neural computation* 9.8 (1997). Publisher: MIT Press, pp. 1735–1780. (Visited on 08/23/2023).

[60] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. url: https://books.google.de/books?hl=en&lr=&id=omivDQAAQBAJ&oi=fnd&pg=PR5&dq=deep+learning&ots=MNV2eosBRS&sig=jN2QwFikq3g_YqU3hJVPEP0XIJ4&redir_esc=y#v=onepage&q=deep%20learning&f=false.

[61] Qiang Wu and Ding-Xuan Zhou. „Analysis of Support Vector Machine Classification“. In: *Journal of Computational Analysis & Applications* 8.2 (2006).

[62] Jianlong Zhou et al. „Evaluating the Quality of Machine Learning Explanations: A Survey on Methods and Metrics“. In: *Electronics* 10.5 (Mar. 4, 2021), p. 593. issn: 2079-9292. doi: 10.3390/electronics10050593. url: https://www.mdpi.com/2079-9292/10/5/593 (visited on 09/11/2023).

[63] Jose Manuel Gomez-Perez, Ronald Denaux, and Andres Garcia-Silva. „Understanding Word Embeddings and Language Models“. In: *A Practical Guide to Hybrid Natural Language Processing: Combining Neural Models and Knowledge Graphs for NLP*. Cham: Springer International Publishing, 2020, pp. 17–31. isbn: 978-3-030-44830-1. doi: 10.1007/978-3-030-44830-1_3. url: https://doi.org/10.1007/978-3-030-44830-1_3.

[64] OpenSSH. *OpenSSH 9.0 release notes*. Retrieved 2022-11-13. Apr. 2022. url: https://www.openssh.com/txt/release-9.0.

[65] Unknown. *How does SSH use both RSA and Diffie-Hellman?* Asked 8 years, 9 months ago; Modified 8 years, 9 months ago; Viewed 22k times; Accessed: 2023-09-21. 2023. url: https://security.stackexchange.com/questions/76894/how-does-ssh-use-both-rsa-and-diffie-hellman.

[66] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. Updated by: 8308, 9141. Network Working Group, Jan. 2006.

[67] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253. Updated by: 6668, 8268, 8308, 8332, 8709, 8758, 9142; Errata Exist. Network Working Group, Jan. 2006.

[68] Girish Venkatachalam. „The OpenSSH Protocol under the Hood“. In: *Linux Journal* 156 (Apr. 2007). url: https://www.ecb.torontomu.ca/~courses/coe518/LinuxJournal/elj2007-156-OpenSSH.pdf.

[69] Oscar M. Guillen et al. „Towards post-quantum security for IoT endpoints with NTRU“. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. 2017, pp. 698–703. doi: 10.23919/DATE.2017.7927079.

[70] Paulo Nunes de Souza and Pavel Gladyshev. „Inference of Endianness and Wordsize From Memory Dumps“. In: *European Conference on Cyber Warfare and Security*. Academic Conferences International Limited. 2017, pp. 619–627.

[71] P. McLaren et al. „Decrypting live SSH traffic in virtual environments“. In: *Digital Investigation* 29 (2019), pp. 109–117. url: https://www.sciencedirect.com/science/article/abs/pii/S1742287619300647.

[72] Tatu Ylonen. *Portable OpenSSH*. Github repositor. Accessed on 25.08.2023. 1995. url: https://github.com/openssh/openssh-portable/.

[73] Auguste Kerckhoffs. „La cryptographic militaire“. In: *Journal des sciences militaires* (1883), pp. 5–38.

[74] SSH Communications Security. *SSH Annual Report 2018*. Annual Report. Accessed: 2023-08-30. SSH Communications Security, 2018. url: https://info.ssh.com/hubfs/2021%20Investor%20documents/SSH_Annual_Report_2018_final.pdf.

[75] W. Yurcik and Chao Liu. „A first step toward detecting SSH identity theft in HPC cluster environments: discriminating masqueraders based on command behavior“. In: *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.* Vol. 1. May 2005, 111–120 Vol. 1. doi: 10.1109/CCGRID.2005.1558542.

[76] *Weak CRC allows packet injection into SSH sessions encrypted with block ciphers*. Accessed: 2023-08-30. Nov. 2001. url: https://www.kb.cert.org.

[77] Core Security Technologies. *SSH Insertion Attack*. https://www.coresecurity.com/core-labs/advisories/ssh-insertion-attack. Archived from the original on 2011-07-08. 2023.

[78] US CERT. *SSH CBC vulnerability. Vulnerability Note VU#958563 - SSH CBC vulnerability*. https://www.kb.cert.org/vuls/id/958563. Archived from the original on 2011-06-22. 2011.

[79] Spiegel Online. *Prying Eyes: Inside the NSA's War on Internet Security*. Spiegel Online. Archived from the original on January 24, 2015. 2014. url: https://www.spiegel.de/international/germany/inside-the-nsa-s-war-on-internet-security-a-1010361.html.

[80] Olivier Bilodeau et al. *Operation WINDIGO*. en. Mar. 2014, p. 69. url: https://web-assets.esetstatic.com/wls/2014/03/operation_windigo.pdf.

[81] Pooneh Nikkhah Bahrami* et al. „Cyber Kill Chain-Based Taxonomy of Advanced Persistent Threat Actors: Analogy of Tactics, Techniques, and Procedures“. In: *Journal of Information Processing Systems* 15.4 (Nov. 2019), pp. 865–889. doi: 10.3745/JIPS.03.0126. url: http://xml.jips-k.org/full-text/view?doi=10.3745/JIPS.03.0126.

[82] ppacher. *honeyssh*. https://github.com/ppacher/honeyssh. GitHub repository. 2017.

[83] Dimitrios Georgoulias et al. „Botnet Business Models, Takedown Attempts, and the Darkweb Market: A Survey“. en. In: *ACM Computing Surveys* 55.11 (Nov. 2023), pp. 1–39. doi: 10.1145/3575808. url: https://dl.acm.org/doi/10.1145/3575808.

[84] Aidan Hogan et al. „Knowledge Graphs“. In: *ACM Comput. Surv.* 54.4 (July 2021). issn: 0360-0300. doi: 10.1145/3447772. url: https://doi.org/10.1145/3447772.

[85] Aidan Hogan et al. „Knowledge Graphs (Extended)“. In: *ACM Computing Surveys* 54.4 (May 2022). arXiv:2003.02320 [cs], pp. 1–37. doi: 10.1145/3447772. url: http://arxiv.org/abs/2003.02320.

[86] Paul Groth et al. „Knowledge Graphs and their Role in the Knowledge Engineering of the 21st Century“. In: *Dagstuhl Reports* 12.9 (2022). Report from Dagstuhl Seminar 22372. Specific usage: pp. 60-72, Subsection "3.2 A Brief History of Knowledge Engineering: A Practitioner's Perspective", pp. 60–120. doi: 10.4230/DagRep.12.9.60.

[87] Marvin Hofer et al. „Construction of Knowledge Graphs: State and Challenges“. In: *arXiv preprint arXiv:2302.11509* (2023). url: https://doi.org/10.48550/arXiv.2302.11509.

[88] Lisa Ehrlinger and Wolfram Wöß. „Towards a Definition of Knowledge Graphs“. In: (2016), pp. 1–4.

[89] Frederick Edward Hulme. *Proverb Lore: Many Sayings, Wise Or Otherwise, on Many Subjects, Gleaned from Many Sources*. E. Stock, 1902, p. 188.

[90] Google. „Introducing the Knowledge Graph: Things, not strings“. In: *Google Blog* (May 2012). Accessed: 2023-06-16. url: https://blog.google/products/search/introducing-knowledge-graph-things-not/.

[91] Michelle Venables. *An Introduction to Graph Theory*. Accessed: 2023-06-12. 2019. url: https://towardsdatascience.com/an-introduction-to-graph-theory-24b41746fabe.

[92] Gianluca Fiorelli. *Best of 2013: No 13 - Search in the Knowledge Graph era*. Accessed: 2023-06-12. 2013. url: https://www.stateofdigital.com/search-in-the-knowledge-graph-era/.

[93] Jackson Gilkey. *Graph Theory and Data Science*. Accessed: 2023-05-25. 2019. url: https://towardsdatascience.com/graph-theory-and-data-science-ec95fe2f31d8.

[94] M.S. Jawad et al. „Adoption of knowledge-graph best development practices for scalable and optimized manufacturing processes“. In: *MethodsX* 10 (2023), p. 102124. issn: 2215-0161. doi: https://doi.org/10.1016/j.mex.2023.102124. url: https://www.sciencedirect.com/science/article/pii/S2215016123001255.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie, dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, October 26, 2023

_____

Rascoussier, Florian Guillaume Pierre