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. Harald Kosch
Zweitprüfer
Prof. Dr. Michael Granitzer

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

November 6, 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 and deep learning models, the study aims to refine feature for embedding techniques and explore innovative methods for effective key detection. 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 [9], enriching both the methodology and the results of the original research while exploring the untapped potential of newly proposed approaches.

The current thesis dives into memory graph modelization from raw binary heap dump files. Each memory graph can support a range of embeddings than can be used directly for model training, through the use of classic ML models and graph neural network. The 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 Prof. Dr. Michael Granitzer and Christofer Fellicious 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, results as well as discussing potential future directions.

Acknowledgements

First acknowledgement goes to Christofer Fellicious, my 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 experimentation.

Another acknowledgements 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 patience, support and invaluable help during the PhDTrack program.
- Elöd Egyed-Zsigmond, PhDTrack coordinator at INSA Lyon, for the subject selection, administrative support and final proofreading.
- All the other PhDTrack students for the great atmosphere, mutual help and the interesting discussions during almost two years.

I cannot forget to mention my many friends from Lyon to Passau and beyond, for their encouragements during this Masterarbeit.

Finally, my last acknowledgements go to my family whose support has been precious throughout the two years of the PhDTrack program.

Contents

List of Figures

List of Tables

Listings

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 an 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 situation presents 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 [74]. Many malwares and botnets «have in common that they have used as attack vector the Secure Shell (SSH) remote access service» [74].

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 enables 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 [83]. The problem lies that those 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» [84].

There are other approaches that allow 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 becomes 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 [9], 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 [45]. It is built on six fundamental principles [44]:

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 generated by *mem2graph*: *mem2graph*: `https://github.com/passau-masterarbeit-2023/memory_graph_gcn`
- **phdtrack_server_scripts**: Scripts for the servers used for computing experiments. 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

The present thesis is organized in a manner that ensures a coherent and logical flow of information, following the standard structure of a Masterarbeit report. The structure is designed to gradually guide the reader from understanding the context and background of the research to the intricacies of the methods employed, and finally to the interpretation of the results. Below is a breakdown of each section:

- **Background Section:** This section serves as an introduction to the research context and establishes the foundation for the thesis. It outlines the previous work and state of the art, providing the reader with an understanding of existing knowledge and identifying gaps that this research aims to address. Key concepts, terminologies, and theories relevant to the study are introduced, setting the stage for the subsequent sections.
- **Methods Section:** This section meticulously describes the methods and approaches employed during the research. From the creation of the dataset to the selection and implementation of machine learning algorithms, this section ensures that the research process is transparent and reproducible.
- **Results Section:** The results' section presents the data obtained from the experiments conducted, outlining both the layout of programs used and the raw results. It provides a factual account of the findings without delving into interpretation or discussion.
- **Discussion Section:** This section offers an analysis and interpretation of the results obtained. It explores the implications of the findings, discusses the limitations of the study, and contextualizes the results within the broader research landscape.
- **Conclusion:** The concluding section succinctly recall the salient points of the thesis. It underscores the contributions made to the field and suggests avenues for future research, providing a fitting closure to the report.

In structuring the thesis in this manner, the intention is to provide the reader with a comprehensive yet accessible insight into the research undertaken all along this year-long project.

2 Background

This section is dedicated to the background information needed to understand the work developed in the thesis. It provides the necessary context for the research, including the problem being solved, why it's important, and the related state-of-the-art and background information. It also includes fundamental concepts and theories, terminology definitions, and a high-level overview of the problem domain. Likewise, it serves as a primer to the rest of the report, providing the necessary context for the research and is intended for readers who may not be experts in the specific area of the research but have some knowledge of the broader field.

2.1 SSH and OpenSSH Implementation

2.1.1 Basics of the Secure Shell Protocol (SSH)

The Secure Shell Protocol, commonly known as SSH, is designed to facilitate secure remote login and other secure network services over insecure networks. SSH has been designed since its inception with security in mind, as a successor of the Telnet protocol, which is not secure, and other «unsecured remote shell protocols such as rlogin, rsh and rexec» [1].

2.1.1.1 SSH design and origin

As stated by the authors of the *SSH Annual Report 2018*, «The founder of SSH, Tatu Ylönen, designed the first version of the SSH protocol after a password-sniffing attack at his university network. Tatu released his implementation as freeware in July 1995, and the tool quickly gained in popularity. Towards the end of 1995, the SSH user base had grown to 20,000 users in fifty countries. By 2000, there were an estimated 2,000,000 users of the protocol. Today, more than 95% of the servers used to power the Internet have SSH installed in them. The SSH protocol is truly one of the cornerstones of a safe Internet.» [75].

SSH is defined in *The Secure Shell (SSH) Protocol Architecture* [67]. It is divided into three major components:

- **Transport Layer Protocol:** This provides server authentication, confidentiality, and integrity. It can also optionally provide compression. Typically, the transport layer runs over a TCP/IP connection but can also be used on top of any other reliable data stream.
- **User Authentication Protocol:** Running over the transport layer, this protocol authenticates the client-side user to the server. Multiple methods of authentication such as password and public key are supported.
- **Connection Protocol:** This multiplexes the encrypted tunnel established by the preceding layers into several logical channels. Channels can be used for various purposes, such as setting up secure interactive shell sessions or tunneling arbitrary TCP/IP ports.

«The client sends a service request once a secure transport layer connection has been established. A second service request is sent after user authentication is complete. This allows new protocols to be defined and coexist with the protocols listed above» [67].

2.1.1.2 SSH keys

For the purposes of this Masterarbeit, a comprehensive understanding of SSH's key exchange and encryption mechanism is important. As outlined in SSHKex [1], the SSH protocol utilizes a key exchange procedure that culminates in a derived master key K and a hash value h. These components are pivotal for encrypting client-server communications and identifying sessions.

During the key exchange process, Diffie-Hellman is employed to negotiate an ephemeral shared key between the client and the server [67] [66]. The Diffie-Hellman key exchange is a method of securely exchanging cryptographic keys over a public channel. Proposed by Whitfield Diffie and Martin Hellman in 1976, this protocol is one of the first practical implementations of public key exchange. The fundamental principle behind Diffie-Hellman is the difficulty of solving discrete logarithm problems [37]. This ephemeral key is then signed by the server using either RSA, DSA, or another suitable signature algorithm (see 2.1.2.2). The signed key confirms to the client that the negotiated key is indeed from the intended server and not an imposter or a middleman, thereby preventing man-in-the-middle (MITM) attacks.

In addition, the host key of the server is used to sign the Diffie-Hellman parameters. This key is not to be confused with the client key listed in the server's `authorized_keys` file, which is used later for client authentication.

The key exchange process results in multiple session keys computed for various purposes:

- **Initialization Vectors:** Key A and Key B are designated for initialization vectors from the client to the server and vice versa.
- **Encryption Keys:** Key C and Key D act as encryption keys for client-to-server and server-to-client communications, respectively.
- **Integrity Keys:** Key E and Key F are utilized to preserve the integrity of data transmitted between the client and server.

This approach provides forward secrecy: if the private key is stolen, it does not compromise the encryption of old sessions. This is because the Diffie-Hellman parameters are ephemeral and discarded once they are no longer needed. Therefore, the only long-lasting keypair is used for authentication purposes.

2.1.1.3 SSH key encryption

These keys are computed using hash functions that take the master key *K* and a hash value *H*, a unique letter (A, B, C, D, E, or F), and the session ID as inputs. This is summarized in „The OpenSSH Protocol under the Hood“: « The equations used for deriving the above vectors and keys are taken from RFC 4253 [68]. In the following, the || symbol stands for concatenation, K is encoded as mpint, K is already a number (hash), "A" as byte and $session_id$ as raw data. Any letter, such as the "A" (in quotation marks) means the single character A, or ASCII 65.

- Initial IV client to server: $HASH(K||H||"A"||session_id)$.
- Initial IV server to client: $HASH(K||H||"B"||session_id)$.
- Encryption key client to server: $HASH(K||H||"C"||session_id)$.

- Encryption key server to client: $HASH(K||H||"D"||session_id)$.
- Integrity key client to server: $HASH(K||H||"E"||session_id)$.
- Integrity key server to client: $HASH(K||H||"F"||session_id)$.

» [69]. Details about the hash function are given in the next section.

The most interesting keys are the encryption keys, as they are used to encrypt the communication between the client and the server. The other keys are used for integrity checks and initialization vectors. Decrypting encrypted SSH communication necessitates either to retrieve these session keys and variables so as to recompute the keys, or to retrieve those keys directly, which is the focus of this Masterarbeit.

2.1.2 OpenSSH Implementation

OpenSSH (OpenBSD Secure Shell) is an open-source implementation written in C of the SSH protocol suite, and it is the most widely used SSH implementation [69]. It is the default SSH implementation on most Linux distributions, and it is also available for Windows. OpenSSH is used for a wide range of purposes, including remote command-line login and remote command execution. It is also used for port forwarding, tunneling, and transferring files via SCP and SFTP either manually or via automated processes, such as backup systems, configuration management tools, and automated software deployment tools.

2.1.2.1 OpenSSH components

OpenSSH is composed of several tools and daemons, including client and server components [73]:

- **ssh:** The basic client program that allows to log into and execute commands on a remote machine.
- **sftp:** An interactive file transfer program that uses SSH to secure the connection.
- **sshd:** This is the SSH daemon that runs on the server. This is used for connecting to a remote machine when using the SSH client from another system.
- **ssh-agent:** The program that holds private keys in memory, so one doesn't have to enter one's passphrase every time.
- **ssh-add:** A program for adding RSA or DSA identities to the authentication agent.
- **ssh-keygen:** A utility for creating and managing SSH keys.
- **ssh-keyscan:** A utility for gathering public SSH host keys from a number of hosts.
- **ssh-keychk:** A utility for checking the validity of SSH keys.
- Several other tools to support the SSH protocol and the OpenSSH implementation.

2.1.2.2 OpenSSH hashing

OpenSSH employs a variety of hash functions and algorithms to secure data, most commonly using SHA1. However, SHA1 is increasingly seen as weak due to its vulnerability to collision attacks [69]. In light of this, the contemporary standard leans towards SHA512. The hash functions are used alongside cipher algorithms like «Advance Encryption Standard (AES) Cipher Block Chaining (CBC), AES Counter (AES-CTR), and ChaCha20» [9]. The Message Authentication Code (MAC) typically uses either MD5 or SHA1 hash algorithms in combination with a secret key. Since cybersecurity and cryptography are constantly evolving, so do SSH and OpenSSH. Depending on the version [69], the available hash options include:

- **ssh-dss:** *(disabled at run-time since OpenSSH 7.0 released in 2015)* SSH-1 version using Digital Signature Algorithm (DSA) from the Digital Signature Standard (DSS). Originally popular but phased out due to vulnerabilities to collision attacks for DSA Key in a 1024-bit modulus. As stated by *Key Exchange (KEX) Method Updates and Recommendations for Secure Shell (SSH)*: «These attacks are still computationally very difficult to perform, but it is desirable that any key exchange using SHA-1 be phased out as soon as possible» [10] [17].
- **ssh-rsa:** *(disabled at run-time since OpenSSH 8.8 released in 2021)* It refers to the use of RSA (Rivest-Shamir-Adleman) encryption algorithm. In the context of SSH-1, this version had to be replaced due to the related to key size issue similar to DSS: «RSA 1024-bit keys have approximately 80 bits of security strength»... «which may not be sufficient for most users.» [10] [22].
- **ecdsa-sha2-nistp256:** *(since OpenSSH 5.7 released in 2011)* Uses the SHA-2 family for hashing and the NIST P-256 curve. It is considered secure and efficient, with an Estimated Security Strength (ESS) of 128 bits [10] [14].
- **ecdsa-sha2-nistp384:** *(since OpenSSH 5.7)* Utilizes the SHA-2 family and the larger NIST P-384 curve for additional security at the cost of performance. It has an ESS of 192 bits [10] [14].
- **ecdsa-sha2-nistp521:** *(since OpenSSH 5.7)* Employs SHA-2 and the even larger NIST P-521 curve for maximal security with an ESS of 256 bits [10]. It is less commonly used due to performance considerations [14].
- **ssh-ed25519:** *(since OpenSSH 6.5 released in 2014)* Known for high security and performance efficiency; employs the Ed25519 elliptic curve with an ESS of 128 bits [10] which is similar to $ecdsa-sha2-nistp256$, and has been more prevalent following the 2013 suspicions of NSA backdoors in NIST curves [13] following the Snowden revelations [11] [12] [16].
- **rsa-sha2-256:** *(since OpenSSH 7.2 released in 2016)* An upgrade from ssh-rsa, using SHA-256 (with ESS of 128 bits) for hashing to improve security without major performance hits [20].
- **rsa-sha2-512:** *(since OpenSSH 7.2)* Similar to $rsa-sha2-256$ but employs SHA-512 for even stronger security, albeit with some performance cost [20].
- **ecdsa-sk:** *(since OpenSSH 8.2 released in 2020)* Security Key-enabled, uses NIST curves and is geared towards modern hardware-based authentication [21].
- **ed25519-sk:** *(since OpenSSH 8.2)* Similar to ssh-ed25519 but integrates hardware-based Security Keys for an additional layer of security [21].

- **NTRU Prime-x25519:** *(since OpenSSH 9.0)* A new, highly secure algorithm focused on post-quantum cryptography, providing future-proof security [70] [65].

These hashes have fixed lengths such that key lengths range between 12 and 64 bytes [9]. Since high-quality random number generation is crucial to ensure that those keys are secure and difficult to predict, it can thus be assumed that those key have a high entropy [72]. This is a crucial assumption as it is the basis for the use of both brute force and machine learning algorithms to predict the presence and location of SSH keys in memory dumps.

The keys generated by these hash functions are pseudo-random numbers stored in the system's RAM. Following the Kerckhoffs' principle: that «a cryptosystem should be secure, even if everything about the system, except the key, is public knowledge», the code for the OpenSSH implementation is open-source and available on GitHub [73]. This allows for the analysis of the code and the identification of the memory structures where the keys are stored.

2.1.3 The state of SSH security

Since its origins, SSH has been developed with cybersecurity in mind, and is generally considered a secure method for remote login and other secure network services over an insecure network. However, as with any technology, it can be exploited if not configured or managed correctly. The protocol is used by system administrators to manage remote systems, and it is also used by automated processes to transfer data and perform other tasks. This makes SSH a valuable target for attackers. In fact, SSH has been a popular target for cyber-attacks. Due to being so prevalent, it is often used by threat actors either as a vector for initial access, as a means to move laterally across a network or as a covered exit for exfiltration of sensitive data [82]. The encrypted nature of its communications makes it an attractive option for attackers, as it can be difficult to detect malicious activity.

2.1.3.1 SSH security issues

Here are some cases where SSH can involve in cyber-attacks, although it's important to note that SSH itself is not inherently insecure:

- **SSH Brute-Force Attacks:** One of the most common types of attacks involving SSH is a brute-force attack, where an attacker tries to gain access by repeatedly attempting to log in with different username-password combinations. These attacks are not sophisticated but can be effective if strong authentication measures are not in place. For instance, the botnet *Chabulo* was used to launch a large-scale brute-force attack «through compromised SSH servers and IoT devices» in 2018 [75].
- **SSH Key Theft:** In some advanced attacks, threat actors have stolen SSH keys to move laterally across a network after initial entry. This allows them to authenticate as a legitimate user and can make detection much more challenging. It can « occur when users have their SSH password or unencrypted keys stolen through a variety of methods (sniffed via a key-logging console program, shoulder-surfed via bad security awareness, poor key management practices, etc.).» [76].
- **Man-in-the-Middle Attacks:** Although SSH is designed to be secure, it can be susceptible to man-in-the-middle attacks if proper verification of SSH keys is not done during the initial connection setup [69].

- **Misconfiguration:** As with any technology, misconfiguration can lead to security issues. For example, leaving default passwords, using weak encryption algorithms, or enabling root login can all make an SSH-enabled system vulnerable [74].

2.1.3.2 SSH vulnerabilities

In cybersecurity, it is generally considered that any system that is connected to the Internet will be attacked at some point. Similarly, it is a common saying that no system is 100% secure. This is true for SSH as well. Although it is a secure protocol, it can be exploited if not configured or managed correctly.

Some vulnerabilities have also been discovered in the protocol itself, although these are rare.

- **SSH-1 Vulnerabilities:** A series of vulnerabilities in the first implementation of SSH were discovered from 1998 to 2001, with its subsequent fixes leading to unauthorized content insertion and arbitrary code execution. SSH-1 had many design flows and is now considered obsolete. [78], [77].
- **CBC Plaintext Recovery:** A theoretical vulnerability discovered in 2008 affecting all versions of SSH, allowing the recovery of up to 32 bits of plaintext from CBC-encrypted ciphertext [79].
- **Suspected Decryption by NSA:** Leaked information in 2014 suggested that the NSA might be able to decrypt some SSH traffic, although the protocol itself was not confirmed to be compromised [80].

2.1.3.3 SSH and cyber-attacks

SSH has been used in many high-profile cyber-attacks and malwares, including the following:

- **Operation Windigo:** This was a large-scale campaign that infected over 25,000 UNIX servers. SSH was one of the vectors used for maintaining control over compromised servers. A report by ESET mentions that the OpenSSH backdoor Linux/Ebury was first discovered in 2011 as a component of the aforementioned operation. «This operation has been ongoing since at least 2011 and has affected high profile servers and companies, including cPanel - the company behind the famous web hosting control panel - and Linux Foundation's kernel.org - the main repository of source code for the Linux kernel» [81].
- **Linux/Hydra:** Initially unleashed in 2008, this malware is a fast login cracker that targets a range of popular protocols including SSH. Hence, SSH is one of its primary vectors to gain initial access to Internet of Things (IoT) devices. Once a device is infected by Linux/Hydra, it joins an IRC channel and initiates a SYN Flood attack [83].
- **Psyb0t:** Discovered in early 2009, Psyb0t is an IRC-controlled malware specifically designed to target devices with MIPS architecture, such as routers and modems. Notably, it was responsible for orchestrating a DDoS attack against the DroneBL service, infecting up to 100,000 devices for this purpose. The malware is equipped to conduct UDP and ICMP flood attacks and employs a brute-force attack mechanism against Telnet and SSH ports. Remarkably, it uses a pre-configured list of 6,000 usernames and 13,000 passwords to perform these attacks [83].

- **Chuck Noris:** Similar to Psyb0t in its objectives and methods, Chuck Noris targets routers and DSL modems, focusing on SoHo (small office/home office) devices. However, unlike Psyb0t, which uses ICMP flood attacks, Chuck Noris deploys ACK flood attacks. The malware carries out brute-force attacks on Telnet and SSH open ports, drawing parallels to the tactics employed by Psyb0t but with the specific variation in flooding techniques [83].

It's worth noting that in many of these cases, SSH was not the initial attack vector but was used at some stage in the attack lifecycle. Properly configured and managed SSH is still considered a secure and robust protocol for remote access and data transfer. In all those situations, a tool monitoring the SSH traffic could have detected the malicious activities and prevented the attack.

2.1.4 The Imperative of SSH Honeypots in Cybersecurity Monitoring

SH (Secure Shell) has become an indispensable protocol for secure communication but can also conceal malicious agents. This reality underscores the urgency for robust monitoring mechanisms capable of identifying suspicious activities in real-time. Among various countermeasures, SSH honeypots have emerged as a particularly effective tool for monitoring and gathering intelligence on potential threats.

An SSH honeypot is a decoy server or service that mimics legitimate SSH services. The primary aim is to attract cybercriminals and study their tactics, thereby offering an active form of surveillance and data collection. Unlike traditional intrusion detection systems, honeypots do not merely identify an attack; they engage the attacker in a controlled environment, enabling detailed observation and logging of the intruder's actions. This allows for the collection of valuable information, such as the attacker's IP address, the tools used, and the techniques employed. This data can then be used to enhance security measures and develop more robust countermeasures [83].

SSH honeypots serve as an invaluable asset in the cybersecurity arsenal, providing not just a reactive but a proactive measure against evolving cyber threats. They can collect actionable intelligence on new hacking methods, malware, and exploitation scripts. This information can be crucial for proactively securing actual production environments. The data collected can also be used to trace back to the origin of the attack, facilitating legal pursuits against the perpetrators. By diverting attackers to decoy servers, honeypots also protect real assets from being targeted, saving both computational resources and administrative effort needed for post-incident recovery.

Popular SSH honeypots include Kippo, Cowrie, and HoneySSH. Cowrie is a fork of Kippo, with additional features such as logging of attacker's keystrokes and file transfer.

- **Kippo:** Kippo is a medium-interaction honeypot that logs the attacker's shell interaction. It specializes in capturing brute force and Telnet-based attacks [83].
- **Cowrie:** Serving as Kippo's successor, Cowrie emulates various protocols including SSH, SFTP, and SCP. It logs events in JSON format, making it particularly useful for detecting brute force and Telnet-based attacks, as well as spoofing attacks [83].
- **IoTPOT:** This IoT-focused honeypot supports multiple CPU architectures and can detect a variety of attacks including brute force, DoS, and sniffing attacks on Telnet, SSH, and HTTP ports [83].
- **HoneySSH:** HoneySSH is a low-interaction honeypot that emulates an SSH server and logs the attacker's IP address, username, and password [85].

- **Sarracenia (SSHKex):** Introduced in 2018, Sarracenia is a high-interaction SSH honeypot that has been enhanced by SSHKex. Instead of «requiring the VM to be paused for every incoming or outgoing packet, which degrades the server performance» [1], SSHKex allows for the extraction of derived SSH session keys. This reduces the performance degradation significantly, as the VM is paused less frequently [1] [15].

These honeypots are useful tools for gathering intelligence on potential threats. However, they are not without their limitations.

2.1.5 Research context and motivation for this Masterarbeit

Security and malware detection are active areas of research, with SSH honeypots being a particularly promising tool for gathering intelligence on potential threats. However, they are not without their limitations. For instance, they are not able to perfectly mimic a real system, such that attackers might be able to detect them „Analysis of SSH Honeypot Effectiveness“.

As explained in „Analysis of SSH Honeypot Effectiveness“: «As attackers become more sophisticated with their ransomware and malware campaigns, there is a significant need for security researchers to assist the greater community by running vulnerable honeypot machines to collect malicious software». Hetzler, Chen, and Khan explain that «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.» They conclude that «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 for security research.» We can extrapolate this conclusion to SSH honeypots, which are also vulnerable to fingerprinting and detection by attackers.

Hence, the need for more advanced SSH honeypots-inspired tools that can leverage data forensic and machine learning techniques so as to be able to use directly a real server as a honeypot, without the need to emulate a system. The current master’s thesis is aligned with this ongoing research (see 2.2), further enhancing the state of SSH honeypots. It aims to develop algorithms, proof of concepts and tools that can extract SSH keys from memory dumps of a real server, and use them to decrypt SSH traffic. This could lead to the development of new tools for SSH monitoring, as discussed in the future work section 6.2.

2.2 Previous Work on OpenSSH key extraction

Now that the necessary context has been established, this section will present the related work in the field of machine learning for memory forensics in the context of OpenSSH. It is divided into two parts. The first part will present the related work in the field of memory forensics, and the second part will present the related work in the field of machine learning for memory forensics.

2.2.1 SSHKex

SSHKex is a research project that aims to address the challenges of analyzing encrypted SSH traffic by leveraging Virtual Machine Introspection (VMI) techniques. Developed by Sentanoe

and Reiser, the project focuses on extracting SSH keys and decrypting SSH network traffic in a stealthy, non-intrusive manner while maintaining evidence integrity [1]. This paper is itself a continuation of the work presented in „Sarracenia: Enhancing the Performance and Stealthiness of SSH Honeypots Using Virtual Machine Introspection“ [15], which introduced Sarracenia, a high-interaction SSH honeypot. It is also related to a range of other research projects and papers [1, section 5.6 and 6].

The SSHKex approach combines standard network traffic capturing methods with dynamic SSH session key extraction. It assumes that the SSH implementation running on the server is known, which is crucial for the key extraction process. The project employs VMI tools like LibVMI and Volatility to gain a complete and untainted view of all guest VM's state information. This allows to efficiently locate SSH session keys in the main memory of a Linux machine.

Here is a summary of the SSHKex methodology for key extraction:

1. **Data Structure Information:** The method leverages detailed knowledge about the data structures used to store the keys. Specific debugging symbols corresponding to the SSH implementation version on the target system provide essential offset values to facilitate the extraction of key material. The structures of interest include `struct ssh`, `struct session_state`, `struct newkeys`, and `struct sshenc`. These structures store a range of information such as IP addresses, ports, session states, and encryption keys.
2. **Tracing OpenSSH Functions:** Function tracing is employed to identify the precise locations of data structures and to extract keys at the right time. The focus is on two key functions: `kex_derive_keys` (which initiates key generation) and `do_authentication2` (which kicks off user authentication).
3. **Breakpoints Injection:** Software breakpoints are intentionally placed in the program execution to facilitate debugging. SSHKex utilizes Virtual Machine Introspection (VMI) to inject these breakpoints at the initial points of the two aforementioned key functions.
4. **Key Extraction:** Upon calling the *kex_derive_keys* function, SSHKex initially stores the address of the *ssh struct*. The actual keys are extracted from memory when the *do_authentication2* function is subsequently called, adhering to the known structures.
5. **Key Indexing:** OpenSSH stores client-to-server and server-to-client keys in distinct indices of the `newkeys` structure. SSHKex extracts keys based on these specific indices.
6. **Handling Multiple Connections:** To manage multiple SSH connections, OpenSSH spawns child processes. SSHKex extends its key extraction strategy to each child process by identifying them through their unique process IDs.

One of the key strengths of SSHKex is its focus on stealthiness, preservation, and evidence integrity. The approach aims to be as unobtrusive as possible, avoiding any modifications to the system under investigation. This is particularly important in forensic contexts, where the integrity of the evidence is crucial [1].

2.2.2 SmartKex

SmartKex is a direct followup project that focuses on the extraction of SSH keys from heap memory dumps. Its primary objective is to automate the process of SSH key extraction from heap memory dumps. The project introduces a machine learning-assisted methodology that

significantly improves the efficiency and accuracy of key extraction compared to traditional brute-force methods. This method is also significantly more straightforward to implement compared to the previous SSHKex approach, which requires detailed knowledge of the SSH implementation and the ability to inject breakpoints into the program execution.

SmartKex discusses two distinct methods for SSH key extraction:

- **Brute-Force Baseline Method:** This is a traditional approach that scans through the heap memory to identify potential keys based on known patterns.
- **Machine Learning-Assisted Method:** This approach uses a Random Forest algorithm trained on a highly imbalanced dataset using SMOTE balancing. The machine learning model is designed to identify SSH keys with high precision and recall rates, but is not exact as compared to the brute-force method since it is based on a probabilistic model.

2.2.2.1 Baseline brute-force method

Here is a summary of SmartKex's brute-force method for SSH key extraction from heap dumps [9]:

1. **Heap Dump Generation:** Heap dump binary files of OpenSSH server process have been generated (ASK HOW) and serves as the input for the key extraction process. The exact process and architecture is not described in SmartKex paper, but we suppose it was done on a *linux-x86_64* architecture.
2. **Data Reduction:** To minimize the heap size, the method removes memory pages that are irrelevant (empty) based on Hamming distance.
3. **Brute-force key search:** Starting from the first byte, a key length of 128 bytes is taken from the heap dump as the potential key. The algorithm iterates over the entire heap, continuously updating the potential key until the heap's end is reached.
4. **Decryption Attempt:** For every potential key, an attempt is made to decrypt network packets. If decryption fails, the process is repeated with a new potential key.

Although the brute-force approach is exact, it is computationally expensive. It performs poorly especially when keys are located at the end of the heap dump [9, section 6.2].

2.2.2.2 SmartKex machine-learning method

The real innovation of SmartKex is its machine learning-assisted methodology for SSH key extraction. At the cost fo exactness, this approach is significantly faster than the brute-force method and has a high degree of accuracy in identifying encryption keys. It also allows for the heap size to be reduced to less than 2% of its original size, further optimizing the extraction process.

Here is a summary of SmartKex's machine learning-assisted method for SSH key extraction from heap dumps [9]:

1. **Heap Dump inputs:** Similarly to the brute-force method, heap dump binary files of OpenSSH also serve as inputs for the key extraction process.

2. **Preprocessing:** The raw heap dump is resized into an $N \times 8$ matrix. High entropy parts of the heap dump, which are likely to be encryption keys, are identified using the logical AND operation on the vertical and horizontal differences of adjacent bytes. This creates an array that flags potential key locations.

3. **Training:** A Random Forest algorithm is trained on 128-byte slices of the preprocessed heap. The dataset is imbalanced, with the slices that contain keys being rare. A stacked classifier approach is used, comprising a high precision classifier and a high recall classifier.

4. **Key Identification:** The machine learning model is used to predict which 128-byte slices of the heap dump are likely to contain encryption keys. These slices are then subjected to a brute-force method to actually extract the keys.

SmartKex is significantly faster than the brute-force method alone and has a high degree of accuracy in identifying encryption keys. It also allows for the heap size to be reduced to less than 2% of its original size, further optimizing the extraction process.

SmartKex has broad applications in the field of cybersecurity, particularly in memory forensics. Its machine learning-assisted methodology can be adapted for other types of sensitive data extraction, making it a versatile tool for researchers and practitioners alike. The project is open-source, with the code available on GitHub[1].

2.2.3 Objectives of the present work

This Masterarbeit can be seen as a direct followup to the paper „SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump“. The present work aims to improve the SmartKex methodology by exploring new machine learning architectures and algorithms. The goal is to improve the accuracy of the machine learning model and to reduce the computational complexity of the overall process.

To do so, this work has significantly broadened the research area by exploring entirely new ways to deal with the dataset by leveraging memory graph representation, feature engineering, new machine learning and deep learning model architectures, and new training strategies. A range of different tools and script, with a focus on code quality and reproducibility with careful packaging using Nix ensure that the present research can be easily extended and reproduced by other researchers.

2.3 Graph-based memory modelization

In the following section, we present important concepts that will be used for the memory modelization of the heap dump.

Because the dataset used is composed of RAW heap dump files from OpenSSH, it is a critical aspect to understand how memory works at a low level point of view. This section aims to provide an in-depth understanding of how memory is managed in C, the language used in the OpenSSH implementation of SSH, for a *linux-x86_64* architecture. We will explore the fundamental concepts of memory management, including the heap and the stack, memory allocation, and the role of

[1] `https://github.com/smartvmi/Smart-and-Naive-SSH-Key-Extraction`

pointers. These concepts will serve later as the foundation for our graph-based approach to memory modelization.

This section will also introduce many graph theory and Knowledge Graph (KG) concepts. We will explore the fundamentals of graph theory, including the definition of a graph, its components, and its properties. We will also discuss the concept of a Knowledge Graph (KG), which is a type of graph that stores information in the form of nodes and edges, and its applications in the field of machine learning.

2.3.1 Defining memory concepts and modelization

Memory management in C is a complex task that requires a deep understanding of the language's features, the operating system's capabilities and the compiler used. In C, memory is primarily managed through two built-in functions: `malloc` (memory allocation) and `free` (memory deallocation). These functions operate on two primary types of memory: the heap and the stack.

- **Heap vs Stack:** The heap is used for dynamic memory allocation, where variables are allocated and freed at runtime. In contrast, the stack is used for static memory allocation, where variables are allocated and deallocated automatically. The stack is faster but has a limited size, while the heap is more flexible but requires manual management to prevent memory leaks.
- **Heap Dump:** A heap dump is a snapshot of the heap's state at a given time. It provides valuable information about the memory layout, active pointers, and data stored in the heap. Analyzing heap dumps can help in debugging memory-related issues and understanding the program's behavior.
- **Memory Addresses:** Each location in memory is identified by a unique memory address. These addresses are usually represented in hexadecimal notation. Note that the address $0x0$ is reserved for the *NULL* pointer, which is used to indicate that a pointer does not point to any memory location.
- **Pointer:** A pointer is a variable that stores the memory address of another variable. It is used to indirectly access the value of the variable it points to. Pointers are used extensively in C, particularly for dynamic memory allocation.
- **Data Structure:** A data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data. In the context of C programming, data structures are declared using the keyword `struct` and are byte aligned. This means that the size of the data structure is always a multiple of the size of the largest member of the structure. Structures can be nested within each other, and pointers can be used to indirectly access the members of a structure. Data structures are often stored in the heap using `malloc`.
- **Malloc headers:** When `malloc` is called, it allocates a block of memory in the heap and returns a pointer to the first byte of the block. The heap manager keeps track of these allocations through metadata, often stored in headers preceding the allocated blocks. These headers contain information such as the size of the allocated block and whether it is free or occupied. Note that the pointer returned by malloc in C points to the first byte of the block of memory that has been allocated for your use, not to the malloc header. The malloc header, is managed internally by the memory allocator and is not exposed to the programmer, but is visible in the heap dump.

2.3.1.1 Endianness

Endianness refers to the byte order used to represent multibyte data types. In a *little-endian* system, the least significant byte is stored first, while in a *big-endian* system, the most significant byte is stored first. Knowing the endianness of the system is crucial for interpreting the content of memory [71].

For instance, the hexadecimal value *0x56343a198000* (taken from *"HEAP_START"* of 3.3) is represented as 550179058774 ($\simeq 5.50e+11$) in decimal basis in a little-endian system, while it is represented as 94782313037824 ($\simeq 9.48e+13$) in a big-endian system.

Little-Endian Conversion The conversion of a hexadecimal number in *little-endian* format to a decimal number is given by the following formula:

$$\text{Decimal} = \sum_{i=0}^{N-1} \left(\text{HexDigit}_i \times 16^i\right)$$

Big-Endian conversion And the conversion of a hexadecimal number in *big-endian* format to a decimal number is given by following formula:

$$\text{Decimal} = \sum_{i=0}^{N-1} \left(\text{HexDigit}_{N-1-i} \times 16^i\right)$$

Here, HexDigit_i is the value of the i-th digit in the little-endian hexadecimal number, and N is the number of digits in the hexadecimal number. Note that HexDigit_i should be converted to its decimal equivalent ('A' becomes 10, 'B' becomes 11, etc.) before performing the calculation.

These formulas will be used later to convert pointer addresses from hexadecimal to decimal format in *mem2graph*.

2.3.1.2 The role of entropy in forensic analysis

Entropy plays a pivotal role in forensic analysis, particularly in the context of memory dumps analysis. It serves as a measure of uncertainty and randomness, which can be crucial for tasks such as endianness detection and identifying encrypted keys in memory.

As defined by Shannon in the realm of information theory, it is a measure of the uncertainty or randomness associated with a set of possible outcomes [71] [25]. In digital applications, when calculated using the logarithm to base 2, entropy represents the amount of bits of information in a message.

Entropy Formula The entropy H of a message is calculated using the formula:

$$H = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Where $p_1, p_2, \ldots, p_n$ are the probabilities of the set of all possible messages [71].

Endianness, as presented before, refers to the byte order used to represent multibyte data types in computer memory [71]. It is necessary to know the endianness of the heap dump to correctly interpret the content of memory, and especially the addresses of potential pointers. In this context,

entropy can be used to infer the endianness of a system by analyzing the distribution of byte values in a memory dump, as presented in „Inference of Endianness and Wordsize From Memory Dumps“ [71].

Entropy is also a key element for encryption key detection, since those should be random byte sequences with high entropy [9]. By examining 8-byte aligned data for high entropy, it is possible to detect potential keys in a heap dump. Techniques such as the calculation of discrete differences and logical operations can further refine this detection as described in „SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump“ [9].

2.3.2 Graphs and Knowledge Graphs

In this project, we (i.e. Clément Lahoche and the author) have built a program called *mem2graph*, a generic tool developed in Rust for efficiency. It is used to convert a RAW memory dump into a graph representation. This graph is technically a directed Edge-labeled heterogeneous property graph, whose concepts are described later. As such, and while it is debatable whether or not *mem2graph* can be seen as building Knowledge Graphs (KGs), it relies on many concepts from the domain, necessitating a proper introduction.

2.3.2.1 Defining Graph Theory concepts

Graph theory is a mathematical field concerned with the study of graphs. It has applications in various fields, including computer science, social sciences, or linguistics. Graphs are used to model pairwise relations between objects, and the study of graphs involves analyzing the properties of these relations. In a few words, a graph is just a collection of nodes and edges.

A graph can be formally defined as: «a pair $G = (S, A)$ where:

- S is a finite set of vertices.
- A is a set of pairs of vertices $(s_i, s_j) \in S^2$.

Graphs can be either directed or undirected. In a directed graph, the pairs (s_i, s_j) are ordered, representing arcs from s_i to s_j. In an undirected graph, the pairs are unordered, representing edges between s_i and s_j » [18]. Let’s introduce some vocabulary to describe graphs:

- **Node (or Vertex)**: A single entity in the graph, often represented as a circle.
- **Edge (or Arc)**: A connection between two nodes, often represented as a line or arrow.
- **Degree**: The number of edges connected to a node.
- **Path**: A sequence of edges that connect two nodes.
- **Cycle**: A path that starts and ends at the same node.
- **Order**: The number of vertices in the graph.
- **Adjacency**: Two nodes are adjacent if there is an edge between them.
- **Loop**: An edge that connects a node to itself.

- **Weight**: A value assigned to an edge.
- **Ancestors (parents) and Descendants (children)**: A node s_i is an ancestor of s_j if there is a path from s_i to s_j. A node s_j is a descendant of s_i if there is a path from s_i to s_j.

Various other terminologies and concepts exist in graph theory, but these are the most important ones for our purposes. For a more in-depth understanding of graph theory, the reader is encouraged to consult the work by Solnon as a quick introduction, [18] or a more in-depth one by West et al. [19].

2.3.2.2 Graphs types

Graphs offer a flexible way to conceptualize, represent, and integrate diverse and incomplete data. Many graph models exist, each with its own advantages and disadvantages, as well as graph properties [2]. Those different types of graphs include:

- **Directed Edge-labelled Graphs (DEL):** The classic graph, set of nodes and edges that connect the nodes with in certain way. RDF is a popular DEL data model.
- **Heterogeneous Graphs:** Each node and edge is assigned one type, allowing for partitioning nodes according to their type, which is useful for machine learning.
- **Property Graphs:** Allows a set of property-value pairs and a label to be associated with nodes and edges. This model is used in Neo4j and offers great flexibility but is harder to handle and query.
- **Graph Dataset:** A set of named graphs, with a default graph with no ID. Useful when working with different sources.
- **Hypergraphs:** Hypergraphs generalize the concept of graphs by allowing edges, known as hyperedges, to connect any number of nodes, rather than just pairs. This means that a single hyperedge can link together two or more nodes, forming a subset of the hypergraph's node set. This feature makes hypergraphs particularly useful for modeling relationships in complex systems where connections are not merely binary. They are widely used in areas such as database design, combinatorial optimization, and complex network analysis, where multi-way relationships are prevalent.

A given graph can have a range of properties that give some insights into its structure. Here are some important properties of graphs:

- **Connected Graph**: A graph is connected if there is a path between every pair of nodes.
- **Disconnected Graph**: A graph is disconnected if there is at least one pair of nodes that are not connected by a path.
- **Cyclic Graph**: A graph is cyclic if it contains at least one cycle.
- **Acyclic Graph**: A graph is acyclic if it does not contain any cycles.
- **Complete Graph**: A graph is complete if there is an edge between every pair of nodes.

[2] Some parts of the following section are directly from a prior work for Seminar 5369S: Knowledge Graphs, written during summer 2023. As of the date of writting and to the best of my knowledge, this work has not been published, and as such, cannot be properly referenced

2.3.2.3 Graph vs Knowledge Graph

A Knowledge Graph (KG) is a specialized form of graph intended to accumulate and convey real-world knowledge. As said before, we are not technically building KGs, but we rely on many concepts from the domain. Research on KG has further accelerated in recent years, and introduced significant improvement to Graph Theory, especially in the practical applications of graph construction and use for Machine Learning or Deep Learning [86]. They have a number of benefits when compared with a relational model or NoSQL alternatives, such as the ability for data to evolve in a more flexible manner, and the capacity to organize data in a way that is not hierarchical. They can represent incomplete information, and does not require a precise schema [86, p.2], which is invaluable in the context of memory analysis, where the structure of the heap is not known in advance.

While all KGs are graphs, not all graphs are KGs. However, the line between the two is often blurry, and the distinction is not always clear. The term "knowledge graph" first appeared in 1973, but really gained popularity through a 2012 blog post about Google's Knowledge Graph [89]. Several definition attempts have been made, but none of them are universally accepted. Below are listed some of the most common definitions of Knowledge Graphs:

- «A graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent potentially different relations between these entities.» [86]
- «A graph of data consisting of semantically described entities and relations of different types that are integrated from different sources. Entities have a unique identifier. KG entities and relations are semantically described using an ontology or, more clearly, an ontological representation.» [88]

In KGs, edges are often labeled and may represent complex relationships like "is a subclass of" or "is married to", allowing for more expressive power. The very nature of KG makes any definition attempt difficult. Indeed, KG is a broad concept that can be applied to many domains, use cases and can have diverse implementations. The definition of KGs is thus very context-dependent, and it's debatable where the line between a graph and a KG is drawn.

For the purpose of this thesis, we won't focus on this distinction and just consider that we deal with memory *graphs*, which are graphs that are not necessarily KGs, but that can be used to represent complex relationships between entities extracted from memory dumps and be leveraged using KG-related techniques for advance tasks like feature engineering, automated embedding, inductive reasoning and learning.

2.3.2.4 Ontologies

An ontology is a formal representation of knowledge within a domain, providing a structured framework for organizing and interpreting information. It consists of a set of concepts, relationships, and rules that define how data is interconnected and how it can be reasoned about. Ontologies are often used to model a domain and support reasoning about entities and their relationships [87].

Ontologies play a crucial role in the development and utility of knowledge graphs. They provide a semantic layer to the knowledge graph, enabling machines to understand the meaning and context of the data. The axioms and rules in an ontology enable automated reasoning,

allowing the knowledge graph to infer new facts from existing data. Ontologies also help in maintaining the quality and consistency of data by enforcing constraints and validation rules. Finally, ontologies enable interoperability by providing a common vocabulary for data exchange and integration. Different popular ontologies exists, such as Web Ontology Language (OWL) or Resource Description Framework (RDF).

By incorporating ontologies, knowledge graphs become more than just a collection of nodes and edges; they become a rich, interconnected web of semantically meaningful information that can be easily queried, analyzed, and extended. We will be referring to concepts that have been inspired by ontologies in the next sections, like *rdf:Bag*.

The `rdf:Bag` container is a part of the Resource Description Framework (RDF) used to represent collections where the order of elements is not significant. Unlike other containers such as `rdf:Seq` and `rdf:Alt`, `rdf:Bag` permits duplicate entries. It can be used to model unsorted collections of resources or literals. An instance of `rdf:Bag` can be represented as a graph where each node connected to the root node represents an item in the collection [8]. For example, consider a root node named `"DataStructure"` with a property `Address` specifying its `malloc header address`.

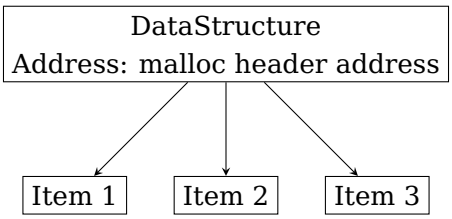


Figure 2.1: Graphical representation of an `rdf:Bag` container.

This small example illustrates how we can represent a container, or a relationship of belonging, using a graph. This is a concept that will be used later in the memory modelization process.

2.3.2.5 Inductive Reasoning and Learning

Inductive reasoning in Knowledge Graphs (KGs) involves techniques like embedding and Graph Convolutional Networks (GCNs) to learn the potential underlying structure of the graph. This is particularly useful for tasks like link prediction, node classification, and clustering.

For the sake of clarity and grouping, we will present the concepts of graph embedding and GCNs in the next sections, but it is important to note that they are not mutually exclusive. In fact, GCNs can be used to generate embeddings, and embeddings can be used as input for GCNs. As such, they are often used together in the context of KGs, or in our case, memory graphs.

2.4 Data preprocessing for Machine Learning

Machine Learning (ML) is a subfield of Artificial Intelligence (AI) that focuses on the development of algorithms that can learn from data and make predictions. It is a powerful tool that has been used to solve a wide range of problems, including image classification, speech recognition, and natural language processing. Before we can apply machine learning algorithms to a dataset, we must first prepare the data by performing various preprocessing steps. In this section, we will discuss the most common data preprocessing techniques, including data cleaning, feature

engineering, and dataset splitting, as well as the importance of feature selection and dimensionality reduction. All those elements are crucial for the development of effective machine learning models for key extraction.

2.4.1 Feature engineering

In the realm of machine learning and data science, *features* refer to individual measurable attributes or characteristics of the phenomena under study. Those features can be of different types, and can be used to predict the value of a target variable.

2.4.1.1 Types of Features

Features can be of different types, depending on the nature of the data. The type of feature determines how it is processed and utilized by the model. The most common types of features include:

- **Numerical Features**: These are quantitative attributes representing measurements like height, weight, or age.
- **Categorical Features**: These are qualitative attributes representing discrete classes or labels, such as gender (Male, Female) or educational level (High School, Bachelor's, Master's).
- **Ordinal Features**: Similar to categorical features but with an inherent order, like ratings on a scale of 1 to 5.
- **Text Features**: These contain textual data and often require special preprocessing steps like tokenization and vectorization.
- **Temporal Features**: These are time-based attributes, such as timestamps, requiring special handling to capture time-dependent patterns.
- **Geospatial Features**: These attributes represent geographical or spatial coordinates.

Features are pivotal for the performance of machine learning models. The quality and pertinence of features can significantly influence the model's capability to discern underlying patterns in the data. Inadequately chosen, or irrelevant features can lead to a poorly performing model, while carefully selected, relevant features can result in a robust and accurate model.

2.4.1.2 The Curse of Dimensionality

The *curse of dimensionality* refers to a set of challenges that arise when dealing with high-dimensional data. As the number of features, or dimensions, in a dataset increases, the volume of the feature space grows exponentially. This exponential growth leads to several issues:

- **Data Sparsity**: In high-dimensional spaces, data points tend to be sparse, making it difficult for algorithms to identify patterns. The sparsity also means that the notion of "distance" becomes less meaningful, which is problematic for algorithms that rely on distance metrics.

- **Computational Complexity**: The exponential increase in volume demands significantly more computational power and memory, making it challenging to process and analyze the data efficiently.
- **Overfitting**: High dimensionality increases the risk of overfitting, where a model learns the noise in the data rather than the actual pattern. Overfit models perform poorly on unseen data.
- **Statistical Significance**: As dimensions increase, the amount of data required to achieve statistical significance also increases exponentially, often making it impractical to collect sufficient data.
- **Visualization and Interpretability**: High-dimensional data are difficult to visualize and interpret, making it challenging to derive intuitive insights.

Due to these challenges, many dimensionality reduction techniques have been developed to transform high-dimensional data into a lower-dimensional form, aiming to preserve as much of the relevant information as possible. These techniques are discussed in the next section.

2.4.1.3 Feature Engineering techniques

The meticulous process of selecting the most relevant features, or constructing new features from existing ones, is known as *feature engineering*. This step can encompass normalization, transformation, and the creation of interaction terms among features. This complex process requires a deep understanding of the domain of study and the datasets, and is often a crucial step in the development of machine learning models [24].

In the context of Machine Learning, features essentially serve as the input variables X that a machine learning model employs to make predictions or inferences about the output variable Y. But not all features are equally informative. Some features may be redundant, irrelevant, or even detrimental to the model's performance. Irrelevant features are those that do not contribute to the predictive power of the model, while redundant features are those that are highly correlated with other features. Feature engineering techniques, can be used to build, transform or eliminate features, thereby reducing the dimensionality of the data and enhancing model performance [23].

- **Scaling and Normalization:** Scaling and normalization are techniques used to transform the features to a similar scale. This is particularly important for algorithms that rely on distance metrics, such as K-Nearest Neighbors (KNN) and Support Vector Machine (SVM). Scaling and normalization can also help accelerate the training process by reducing the number of iterations required for the model to converge. Some common techniques include min-max scaling, z-score normalization, and log transformation.
- **Feature Extraction**: This technique involves transforming the original set of features into a new set of features, which is usually of lower dimensionality. The new features are often combinations of the original features and aim to capture as much of the information in the original data as possible. Many methods exits, like Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA) and t-distributed Stochastic Neighbor Embedding (t-SNE) are often employed to transform high-dimensional data into a lower-dimensional form are commonly used for feature extraction [23].
- **Feature Selection**: Unlike feature extraction, feature selection aims to pick a subset of the most important features from the original set, without changing them [23]. The goal is to

remove irrelevant or redundant features that do not contribute significantly to the model's performance. Techniques like Recursive Feature Elimination (RFE) and using importance scores from tree-based algorithms like Random Forest are popular methods for feature selection.

Both techniques have their own advantages and disadvantages, and the choice between the two often depends on the specific requirements of the task at hand. Feature extraction is generally more suitable when the original features do not have much interpretability to begin with, or when transforming features can lead to a more compact and effective representation. On the other hand, feature selection is often preferred when it is important to maintain the interpretability of the features, or when computational efficiency is a concern [24].

2.4.1.4 Evaluating Features with Correlation Tests

To assess the quality of features, various statistical measures can be employed. Correlation tests are statistical tests that measure the strength and direction of the relationship between two variables. Pearson, Kendall, and Spearman correlation coefficients are commonly used to quantify the linear or monotonic relationship between each feature and the target variable [27]. A high absolute value indicates a strong relationship, aiding in feature selection.

- **Pearson Correlation**: Measures the linear relationship between two variables. It ranges from -1 to 1, where -1 indicates a strong negative linear correlation, 1 indicates a strong positive linear correlation, and 0 indicates no linear correlation.
- **Kendall's Tau**: A non-parametric test that measures the strength and direction of a monotonic relationship between two variables.
- **Spearman's Rank**: Also a non-parametric test, it assesses how well an arbitrary monotonic function can describe the relationship between two variables without making any assumptions about the frequency distribution.

These techniques are useful for evaluating the relationship between each feature and allows generating correlation matrices, which can be used to identify redundant features. It's also possible to evaluate each feature independently through univariate feature selection techniques. In Python's Scikit-learn library [26], methods like *F-test* and the *p-value* are often used for this purpose. "

- **F-test value**: Measures the linear dependency between the feature variable and the target. A higher F-test value indicates a more useful feature.
- **p-value**: Indicates the probability of an F-test value this large arising if the null hypothesis is true. A smaller p-value suggests rejecting the null hypothesis, making the feature significant.

In summary, features are the foundational elements of any machine learning model. The quality of these features, along with how they are processed and utilized, can markedly impact the model's performance. The significance of feature engineering cannot be overstated. Properly engineered features can drastically reduce modeling errors, leading to more accurate and reliable predictions. It serves as a bridge between raw data and predictive models, ensuring that the models are fed with the most relevant and informative features.

2.4.2 Embeddings

Embeddings are low-dimensional vector representations of high-dimensional objects. They are often used to capture complex relationships between objects and are particularly useful for machine learning tasks like clustering, classification, and link prediction. Embeddings are widely used in the field of natural language processing (NLP) to represent words, sentences, and documents [64]. In recent years, they have also been applied to graphs to learn node representations [86]. In this section, we will discuss the concept of embeddings and explore some common techniques for creating them.

2.4.2.1 Embedding Creation vs Feature Engineering

Both embedding creation and feature engineering are techniques used to transform data for machine learning models. However, they are different in terms of their goals and how they are achieved.

- **Embedding Creation**: Embedding creation is the process of learning a low-dimensional vector representation of a high-dimensional object. This involves mapping discrete objects, such as words in Natural Language Processing (NLP) or nodes in a graph in our case, to vectors of continuous values in a lower-dimensional space [64]. The goal is to capture the semantic or structural relationships between these objects. Specialized algorithms like Word2Vec for word embeddings or Node2Vec for graph embeddings are often used.

- **Feature Engineering**: As discussed before, it is a more general practice that involves creating new features or modifying existing ones to improve the performance of a machine learning model. While feature engineering can include creating embeddings, it also encompasses a wide range of other techniques like normalization, transformation, outlier detection, and handling missing values.

In general, feature engineering is a more manual process, while embedding creation is more automated. Feature engineering requires domain knowledge and understanding of the problem, while embedding creation can be done using a variety of machine learning techniques.

Thus, embedding creation is a specialized form of feature engineering aimed at mapping discrete objects to continuous vectors of numbers, usually for capturing complex relationships. Feature engineering, on the other hand, is a broader practice that can involve a variety of techniques, including but not limited to embedding creation.

2.4.2.2 Embeddings for graphs

Graph embedding techniques aim to map nodes and edges in a graph to vectors in a low-dimensional space [86]. The primary goal is to preserve the graph's structural properties, such as node connectivity and community structure, in the embedded space. These vectors can then be used for various machine learning tasks like clustering, classification, and link prediction. A quite complete overview of the different techniques can be found in „Knowledge Graphs“ [86, Section 4.2]. Here are some common and advanced techniques:

Translational Models The first type of graph embeddings techniques is based around using transactional models that interpret edge labels as transformations from subject nodes to object nodes „Knowledge Graphs“.

- **TransE:** is one of the earliest and most straightforward translational models. It represents entities as points in a vector space and relations as translations between these points. The primary idea is that for a valid triple (h, r, t), the equation $h+r = t$ should hold, where h stands for the head entity, r represents the relation, and t is the tail entity. This model is simple and computationally efficient but is limited in its ability to capture complex relationships [86].
- **TransH:** TransH extends TransE by introducing relation-specific hyperplanes. This allows the model to capture more complex relationships by projecting the entity embeddings onto these hyperplanes before performing translations [86].
- **TransR:** TransR goes a step further by not only introducing relation-specific hyperplanes but also relation-specific translations. This allows for a more flexible representation of relations, accommodating various types of complexities [86].
- Other improvements include *TransD* or *MuRP*, which shows that research in this domain is still very active.

Tensor Decomposition Models Tensor decomposition models represent entities and relations as vectors or matrices in a low-dimensional space. These models are based on the assumption that the relationship between entities can be represented by a bilinear function. They are computationally efficient and can capture complex relationships, but they are also limited in their ability to model asymmetric and reflexive relations.

- **RESCAL:** RESCAL employs a bilinear model where each relation is represented by a full-rank matrix. This allows for capturing asymmetric and reflexive relations but at the cost of increased computational complexity [86].
- **DistMult:** DistMult simplifies RESCAL by assuming that the relation matrices are diagonal. This reduces the number of parameters and computational complexity, making it more scalable [86] [87].
- **ComplEx:** ComplEx extends DistMult by introducing complex-valued embeddings. This allows the model to capture asymmetric relations effectively while maintaining computational efficiency [86] [87].

Neural Models Neural models employ neural networks to learn the features of entities and relations. This provides a more flexible and adaptive approach to graph embeddings.

- **ConvKB:** ConvKB employs a convolutional neural network to automatically learn the features of entities and relations. It is technically a translational model as introduced before, but uses a convolutional layer to capture the interactions between entities and relations. This provides a more flexible and adaptive approach to graph embeddings [33].
- **RotatE:** RotatE uses complex rotations in the embedding space to model relations. This neural model captures the semantics of relations in a more expressive manner [87].

- **SDNE (Structural Deep Network Embedding)**: SDNE employs a deep autoencoder to learn complex and non-linear node embeddings while preserving first-order and second-order proximities. It is particularly effective for capturing intricate patterns and structures in the graph [34].
- **R-GCN:** Relational Graph Convolutional Networks (R-GCNs) combine the strengths of GCNs and traditional embedding methods to capture both topological and semantic information [35]. A more recent study by Degraeve et al. argues that the main contribution of R-GCN lies in its message passing paradigm rather than the learned weights. This paper introduces a variant called Random R-GCN (RR-GCN) [48].
- **ConvE:** ConvE employs convolutional layers to capture local and global interactions between entities and relations, offering a more expressive representation [87].

Language Models Language models utilize pre-trained language models to enrich the embeddings with contextual information. This approach leverages the recent developments of language models to capture the semantics of entities and relations.

- **BERT for KGE:** Utilizing pre-trained BERT models, this approach leverages the power of language models to enrich the embeddings with contextual information [36].
- **BART KGE:** Bidirectional and Auto-Regressive Transformers (BART) is a denoising autoencoder that can be used for various NLP tasks. This approach utilizes BART to learn entity and relation embeddings. The paper introducing this also compares other LLM like GPT-2 [38].
- Due to the recent developments in NLP, this domain is still very active, and new approaches are being developed regularly.

2.4.2.3 Word Embeddings

Word embeddings are vector representations of words in a low-dimensional space. They are often used as input for machine learning models in natural language processing (NLP) tasks like text classification, sentiment analysis, and machine translation. Word embeddings are typically learned from large text corpora using unsupervised learning techniques like Word2Vec and GloVe. These embeddings can then be used to capture semantic relationships between words and phrases, which is particularly useful for NLP tasks [64].

- **Word2Vec**: Word2Vec is a popular algorithm for learning word embeddings from text data. It employs a shallow neural network to learn the embeddings and is often used as a pre-processing step for NLP tasks [87].
- **GloVe**: Global Vectors for Word Representation (GloVe) is another popular algorithm for learning word embeddings. It is based on the co-occurrence matrix of words and utilizes matrix factorization to learn the embeddings [87].

Graph Embeddings Graph embeddings are vector representations of nodes in a graph.Graph embeddings are typically learned using unsupervised learning techniques like Node2Vec and DeepWalk. These embeddings can then be used to capture structural relationships between nodes, which is particularly useful for graph analytics tasks [86].

- **One-Hot Encoding**: This is a simple technique that represents each node as a vector of 0s and 1s, where the length of the vector is equal to the number of nodes in the graph. The vector contains a 1 at the index corresponding to the node and 0s everywhere else. As is, this method is not really suitable for large graphs as it results in a high-dimensional and sparse matrix representation [87].

- **Node2Vec**: This algorithm learns continuous feature representations for nodes by optimizing a neighborhood-preserving objective. It employs biased random walks and uses the Skip-gram model to generate embeddings. Node2Vec is particularly effective for capturing local structures and can be fine-tuned for specific tasks [39].

- **DeepWalk**: Similar to Node2Vec, DeepWalk uses random walks to generate node sequences. It employs the Skip-gram model from natural language processing to learn embeddings. Unlike Node2Vec, it does not use biased walks, making it more suitable for capturing global structures [49].

- **Spectral Clustering**: This technique is based on the spectral theory of graphs. It utilizes the eigenvalues and eigenvectors of the Laplacian matrix of the graph to find an optimal embedding. Spectral Clustering is particularly useful for community detection and can capture the global structure of the graph [55] [57].

- **LINE**: Large-scale Information Network Embedding (LINE) aims to preserve both local and global network structures. It optimizes two objectives: first-order and second-order proximities between nodes. LINE is scalable and can handle large graphs efficiently [40].

- **Graph Factorization**: This method directly factorizes the adjacency matrix of the graph to learn node embeddings, making it computationally efficient but less capable of capturing complex structures. It is often used for large-scale graphs where computational resources are limited [87].

2.4.3 Other preprocessing techniques for Machine Learning

When working with real-world data and datasets, it is common to find issues like missing values, outliers, and imbalanced classes that can adversely affect the performance of machine learning models. In this section, we will discuss some common preprocessing techniques used to improve the quality of data and ensure that the models can learn effectively.

2.4.3.1 Data Cleaning

Data cleaning is the process of detecting and correcting errors in the data. It is a crucial step in data preprocessing and involves various techniques like outlier detection, handling missing values, and data normalization. Data cleaning is necessary to ensure that the data is accurate and consistent, which is essential for machine learning models to learn effectively.

- **Outlier Detection**: Outliers are data points that deviate significantly from the rest of the dataset. They can be caused by errors in data collection or genuine variations. Outliers can skew the model's learning and should be identified and handled appropriately, either by removal or transformation.

- **Handling Missing Values**: Missing data can lead to biased or incorrect model training. Techniques for handling missing values include imputation, where missing values are replaced with statistical measures like mean, median, or mode, and deletion, where rows with missing values are removed.
- **Data Normalization**: Features with different scales can affect the performance of machine learning algorithms. Normalization rescales the features to a standard range, usually [0, 1], or transforms them to have a mean of 0 and a standard deviation of 1.
- **Encoding Categorical Variables**: Many machine learning algorithms require numerical input and output variables. Categorical variables are converted to numerical format through techniques like one-hot encoding or label encoding.
- **Text Cleaning**: In natural language processing tasks, text data may require cleaning to remove irrelevant characters, correct typos, or standardize text format.
- **Duplicate Removal**: Duplicate entries can bias the model and should be identified and removed from the dataset.
- **Feature Engineering**: While not strictly data cleaning, feature engineering involves transforming existing features or creating new ones to improve model performance.

2.4.3.2 Dataset splitting and sampling

One other typical step is the division of the dataset into training and testing sets. This separation is crucial for evaluating the generalization performance of a model. The training set is used to train the model, while the testing set is used to evaluate its performance on unseen data. Failing to separate these sets can lead to overfitting, where the model performs well on the training data but poorly on new, unseen data.

Various techniques exist for dataset splitting and sampling, each with its own advantages and disadvantages:

- **Random Split**: The dataset is randomly divided into training and testing sets based on a given ratio, such as 70% for training and 30% for testing. This method is simple but may result in imbalanced classes in the splits.
- **Stratified Split**: Similar to random split, but ensures that the distribution of classes is the same in both training and testing sets. This is particularly useful for imbalanced datasets.
- **k-Fold Cross-Validation**: The dataset is divided into ’k’ subsets or "folds." The model is trained on k-1 folds and tested on the remaining fold. This process is repeated k times, each time with a different fold as the testing set. The average performance metric is used for evaluation.
- **Leave-One-Out Cross-Validation (LOOCV)**: A special case of k-Fold Cross-Validation where k is equal to the number of data points. Each data point is used once as the test set while the remaining points form the training set.
- **Bootstrapping**: Random samples are drawn with replacement from the dataset to create multiple training sets. The model is trained and tested on these sets, and the average performance is calculated.

2.4.3.3 Dealing with Imbalanced Datasets

Imbalanced datasets are those where the classes are not represented equally. This is a common issue in machine learning, especially in classification problems. An imbalanced dataset can lead to a biased model that may not effectively predict the minority class. Therefore, it is crucial to address this issue during data preprocessing.

- **Why it is Necessary**: In imbalanced datasets, machine learning algorithms tend to be biased towards the majority class, ignoring the minority class. This results in poor classification performance for the minority class, which is often the class of interest in problems like fraud detection, medical diagnosis, etc.
- **Oversampling**: This involves adding more copies of the minority class. Oversampling can be random with replacement, or it can involve generating synthetic samples.
- **Undersampling**: This involves removing some of the samples of the majority class. This method is generally not preferred as it can lead to loss of data.
- **SMOTE (Synthetic Minority Over-sampling Technique)**: SMOTE is an oversampling method that creates synthetic samples in the feature space. It selects two or more similar instances (using a distance measure) and perturbing an instance one at a time by a random amount within the difference to the neighboring instances.
- **ADASYN (Adaptive Synthetic Sampling)**: Similar to SMOTE, but it uses a weighted distribution for different minority class examples according to their level of difficulty in learning.
- **Cost-sensitive Learning**: This involves modifying the algorithm to increase the weight of the minority class during training, thereby making the algorithm more sensitive to it.
- **Ensemble Methods**: Methods like Random Forest and Gradient Boosting can be used with techniques like bagging and boosting to handle imbalanced datasets effectively.
- **Resampling Methods**: These involve randomly partitioning the data into subsets, balancing each subset, and then aggregating the results.

To conclude this section, data preprocessing is a crucial step in machine learning. The number of concepts and techniques discussed here is by no means exhaustive, which further highlights the importance and complexity of data preprocessing.

2.5 Machine Learning and Deep Learning

In the preceding sections, we have discussed the importance of data preprocessing and feature engineering for machine learning. In this section, we will discuss the machine learning pipeline and explore some common machine learning algorithms. We will also discuss deep learning and neural networks especially in the context of graphs, since they have gained popularity in recent years due to their superior performance on many tasks.

2.5.1 Machine Learning

Machine learning is a cornerstone in the field of artificial intelligence and has been instrumental in driving many of today's technological and scientific breakthroughs. From natural language processing to computer vision, machine learning algorithms play a critical role in making sense of large and complex data sets.

2.5.1.1 What is Machine Learning

Machine learning is a subfield of artificial intelligence that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. This learning process is based on the recognition of complex patterns in data and the making of intelligent decisions based on them [28].

The machine learning pipeline typically involves several steps including some pre-steps like data collection, preprocessing, feature extraction that we have already discussed before, usually followed by model training, evaluation, and deployment. Algorithms are trained on a dataset, and the learned patterns are used to make predictions or decisions without human intervention. ML algorithms can be broadly classified into three categories [28]:

- **Supervised Learning**: Algorithms are trained on labeled data, and the aim is to make predictions or map inputs to outputs.
- **Unsupervised Learning**: Algorithms are trained on unlabeled data, focusing on the underlying structure or distribution in the data.
- **Reinforcement Learning**: Algorithms learn to perform an action from state to state to maximize some type of reward or objective function.

In the following, we will focus on supervised and unsupervised learning, as they are the most relevant to our work.

2.5.1.2 Model Evaluation

Evaluating the performance of a machine learning model is crucial for understanding its effectiveness and suitability for a given task. It is not a trivial task as it depends on various factors like the type of data, the problem at hand, and the model itself [63]. In this section, we will discuss some common evaluation metrics for classification and regression models.

Precision Precision is the ratio of correctly predicted positive observations to the total predicted positives. The formula for precision is:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Recall Recall is the ratio of correctly predicted positive observations to all the observations in the actual class. The formula for recall is:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

F-1 Score The F-1 Score is the weighted average of Precision and Recall, and it ranges from 0 to 1. The formula for the F-1 Score is:

$$\text{F-1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Accuracy Accuracy is the ratio of correctly predicted observations to the total observations. The formula for accuracy is:

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Observations}}$$

These are the most common metrics used for evaluating classification models. Other metrics like the Area Under the Curve (AUC) and the Receiver Operating Characteristic (ROC) curve are also used for evaluating binary classification models. For regression models, metrics like Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE) are commonly used [26].

2.5.2 Machine Learning Models for Binary Classification

Binary classification is a supervised learning task where the goal is to predict a binary outcome, i.e., one of two possible classes (0 or 1). This is a truly classical task for a ML model, with a range of applicable algorithms. As such, we rely on Python's Scikit-learn library for implementation.

Logistic Regression A specialized form of regression tailored for predicting binary outcomes. It employs the logistic function to map predicted values between 0 and 1. While straightforward and interpretable, its performance may be limited on complex, non-linear data [29].

The logistic function is defined as:

$$P(y = 1|x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

Where β_0 is the intercept and β_1 is the coefficient for the predictor variable x.

Decision Trees These are versatile models used for both classification and regression. They partition the feature space into regions, making decisions at each node. While easy to visualize, they are susceptible to overfitting [30].

Decision Trees use metrics like Gini impurity or entropy to make splits:

$$\text{Gini}(T) = 1 - \sum_{i=1}^{C} p_i^2$$

Where T is a node and p_i is the proportion of class i instances among the training instances in node T.

Random Forest An ensemble technique that aggregates predictions from multiple decision trees. Known for its robustness and ability to handle large, high-dimensional data [31].

The final prediction is an average or majority vote from all trees:

$$\hat{y} = \frac{1}{n} \sum_{i=1}^{n} \hat{y}_i$$

Where $\hat{y}$ is the final prediction and $\hat{y}_i$ is the prediction from the i^{th} tree.

Support Vector Machines (SVM) Effective for both classification and regression, SVMs find the hyperplane that best separates the data into classes. They excel in high-dimensional spaces [62].

The objective is to maximize the margin between classes:

$$\max_{w,b} \frac{2}{\|w\|}$$

Where w is the weight vector and b is the bias term.

k-Nearest Neighbors (k-NN) An instance-based algorithm that classifies a new point based on the majority class among its 'k' nearest neighbors [59].

The distance between points is often calculated using Euclidean distance:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Where $d(x, y)$ is the distance between points x and y, and n is the number of dimensions.

2.5.3 Deep Learning

Deep learning offers a powerful set of tools for automatically generating embeddings from the graph representation of heap dumps. Leveraging neural networks, we can build custom models using PyTorch to perform binary classification tasks, such as predicting key nodes in a heap dump. In this section, we will explore various neural network architectures that are well-suited for this task.

Neural Networks serve as the backbone of deep learning models [61]. They consist of interconnected nodes or "neurons" organized into layers. Deep learning extends this architecture by employing multiple hidden layers, enabling the model to learn complex representations from data. This is particularly useful when dealing with graph-based heap dump representations, where the relationships between nodes can be intricate [87].

Choosing Deep Learning models is dependent on the task. As such, we need to present the different types of neural networks and their advantages and disadvantages. Below are some common neural network architectures:

2.5.3.1 Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNN) are particularly effective for sequence-based data. In the context of heap dumps, if the memory addresses or keys exhibit some form of sequential pattern, RNNs can be employed to capture these temporal dependencies for binary classification [61, p. 10.2].

- **Advantages:**
  - Good at capturing short-term dependencies in sequence data.
  - Relatively simpler architecture.
- **Disadvantages:**
  - Struggles with long-term dependencies due to the vanishing gradient problem.
  - May require more data for effective training.

2.5.3.2 Long Short-Term Memory (LSTM)

LSTMs are an extension of RNNs designed to capture long-term dependencies, making them suitable for more complex sequences [60].

- **Advantages:**
  - Effective in capturing long-term dependencies.
  - Less susceptible to the vanishing gradient problem.
- **Disadvantages:**
  - More complex and computationally intensive than RNNs.
  - May require fine-tuning of hyperparameters.

2.5.3.3 Gated Recurrent Units (GRU)

Gated Recurrent Units (GRU) offer a compromise between the simplicity of RNNs and the power of LSTMs. They are effective in capturing both short-term and long-term dependencies but with a less complex architecture [58].

- **Advantages:**
  - Simpler architecture compared to LSTM.
  - Efficient in capturing long-term dependencies.
- **Disadvantages:**
  - May not perform as well as LSTMs for very complex sequences.
  - Still more computationally intensive than basic RNNs.

2.5.3.4 Convolutional Neural Networks (CNN)

Though traditionally used in image processing, CNNs can also be adapted for sequence data like heap dumps. They are excellent at identifying spatial hierarchies or patterns in the data [61] [32].

- **Advantages:**
  - Highly effective in identifying local patterns.
  - Less prone to overfitting due to pooling layers.
- **Disadvantages:**
  - May not capture global dependencies as effectively as RNNs or LSTMs.
  - Requires a fixed-size input.

2.5.3.5 Graph Convolutional Networks (GCN)

Graph Convolutional Networks (GCNs) are a specialized form of neural networks designed to work directly with graphs [87]. They are particularly useful for our task as they can automatically generate embeddings from the graph representation of heap dumps. These embeddings can then be used for binary classification to predict key nodes.

- **Advantages:**
  - Capable of capturing both local and global graph structures.
  - No need for manual feature extraction from graphs.
- **Disadvantages:**
  - May require fine-tuning and a well-defined graph structure.
  - Computationally intensive for large graphs.

Several Python libraries can be used to implement GCNs for our specific task. Some notable ones are PyTorch Geometric, Spektral, and DGL (Deep Graph Library). These libraries offer pre-built GCN layers and various utilities to facilitate the embedding and binary classification of heap dump graphs.

2.5.4 Graph Neural Networks

A Graph Neural Network (GNN) constructs a neural network that mirrors the structure of the underlying data graph. In this architecture, nodes are linked according to their relationships in the data graph. The model is trained in a supervised fashion to map input features of nodes to their corresponding output features [87]. These output features can either be manually annotated or sourced from a knowledge graph.

Unlike traditional knowledge graph embeddings, GNNs offer the advantage of end-to-end supervised learning tailored for specific tasks.

With a dataset of labeled examples, GNNs can classify individual nodes or even entire graphs. The challenge lies both in how to convert a given graph to a format that can be fed into a neural network (see 2.4.2.2) and how to design a neural network that can effectively learn from the graph data. Note that the distinction between the two is not always clear since two problems are often tackled together. As such, the present classification is not exhaustive and arbitrary, even though it is based on the literature and meta-analysis of the field, like „Knowledge Graphs (Extended)“ [87] and „A comprehensive survey on graph neural networks“ [51].

GNNs have been employed in various applications, ranging from traffic prediction, recommender systems, and software verification [51]. Remarkably, GNNs can also serve as substitutes for conventional graph algorithms. For instance, they have been used to identify central nodes in knowledge graphs through supervised learning [87]. The following sections delve into two specific types of GNNs: Recursive GNNs and Convolutional GNNs.

2.5.4.1 Recursive Graph Neural Networks (RecGNNs)

Recursive Graph Neural Networks (RecGNNs) serve as the foundational approach to graph neural networks [87]. The model operates by passing messages between neighboring nodes to recursively compute results. The framework learns the functions that generate the expected output based on a training set of labeled nodes. Scarselli et al. [54] proposed a seminal GNN model that uses feature vectors for nodes and edges, along with state vectors for nodes. Two parametric functions, the transition function and the output function, are used to update the state vectors and compute the final output for nodes, respectively. These functions are applied recursively until a fixpoint is reached. The model is highly flexible and can be adapted in various ways, such as defining neighboring nodes differently or using distinct parameters for each node [54].

Learning Process in GNNs The learning process in GNNs can be divided into three steps: input computation, node state update, and output computation. These steps are repeated until a fixpoint is reached, and the final output is computed for each node. Essentially, this involves finding the optimal parameters w such as for φ_w to best approximate the data in the learning dataset $\mathcal{L}$.

The learning task can be formulated as the minimization of a quadratic cost function e_w through iterative gradient descent [54]:

$$e_w = \sum_{i=1}^{p} \sum_{j=1}^{q_i} (t_{i,j} - \varphi_w(G_i, n_{i,j}))^2$$

Where:

- $\mathcal{G}$: The set of graphs.
- $\mathcal{N}$: The subset of nodes.
- $\mathcal{D} = \mathcal{G} \times \mathcal{N}$: The set of pairs of a graph and nodes.
- $G_i = (N_i, E_i) \in \mathcal{G}$: The graph.
- N_i: The set of nodes in graph G_i.
- E_i: The set of edges in graph G_i.
- $n_{i,j} \in N_i$: The j^{th} node of the graph G_i.
- $q_i \leq |N_i|$: The number of supervised nodes in the graph G_i.
- $p \leq |\mathcal{G}|$: The number of graphs in the learning set.
- $\mathcal{L} = \{(G_i, n_{i,j}, t_{i,j})\}$: The learning set.
- $t_{i,j} \in \mathbb{R}^m$: The target output for node $n_{i,j}$.
- m: The number of outputs.
- $\mathbb{R}^m$: The m-dimensional Euclidean space.
- $\varphi_w : \mathcal{D} \to \mathbb{R}^m$: The function that maps the input to the output.
- e_w: Error function.

For graph-focused tasks, a special node is used for the target (t), whereas for node-focused tasks, supervision can be performed on every node. More information about the learning process can be found in [54].

2.5.4.2 Convolutional Graph Neural Networks (ConvGNNs)

As introduced before, Graph Convolutional Networks extend the concept of convolution from images to graphs. «The core idea in the image setting is to apply small kernels (aka filters) over localized regions of an image using a convolution operator to extract features from that local region.» [87]. They are designed to work with non-Euclidean data and are particularly useful for semi-supervised learning tasks on graphs. GCNs aim to learn a function that maps nodes to a low-dimensional space while considering their local neighborhood and features.

Convolution filters A Convolutional Neural Network (CNN) is a type of neural network that uses convolutional filters to extract features from images. These filters are applied to local regions of the image to capture spatial patterns [42]. The same concept can be extended to graphs, where the convolution filters are applied to local regions of the graph to capture structural patterns.

The convolution operation on an image is historically defined as follows [47] [56]:

$$y_{i,j} = \sum_{k=-m}^{m} \sum_{l=-n}^{n} x_{i+k,j+l} \times w_{kl}$$

Where:

- $y_{i,j}$: The output value at the coordinate (i, j) of the convolution output.
- $x_{i,j}$: The input value at the coordinate (i, j) of the input image.
- w_{kl}: The weight value (coefficient) of the kernel at the coordinate (k, l).
- m: The kernel width.
- n: The kernel height.

Similar to this, ConvGNNs implement the transition function using convolutions. One of the challenges in ConvGNNs is defining regions of a graph, as nodes in a graph may have varying numbers of neighbors. Solutions to this problem involve using spectral or spatial representations of graphs [57] [87]. An alternative approach employs attention mechanisms to learn the most important features [41]. Unlike RecGNNs, ConvGNNs apply a fixed number of convolutional layers and can use different kernels/weights at each distinct step.

Here are some common ConvGNN architectures. I recommend reading „Beyond low-pass filtering: Graph convolutional networks with automatic filtering“ [52] for a more in-depth discussion of these models:

- **Vanilla GCN**: The basic GCN model consists of an input layer, one or more hidden layers, and an output layer. Each layer is associated with a graph convolution operation that updates the node features based on their neighbors [87].
- **GraphSAGE (Graph Sample and Aggregation)**: This model generalizes the GCN framework by allowing various aggregation functions like mean, LSTM, and pooling to combine information from a node’s neighbors [50].
- **ChebNet**: ChebNet uses Chebyshev polynomials to generalize the convolution operation in the spectral domain. This allows the model to capture a broader range of graph structures [51].
- **GAT (Graph Attention Networks)**: GAT introduces attention mechanisms into GCNs, enabling the model to weigh neighbors differently when aggregating information [41].
- **MoNet**: This model employs a mixture model to generalize the convolution operation, making it capable of handling graphs with diverse structures [53].

- **Graph Isomorphism Network (GIN)**: GIN is designed to capture the isomorphism between different graphs, making it powerful for tasks like graph classification [52].
- Research on the topic is still ongoing, and a lot of new architectures are being proposed: Cluster-GCN, TAGCN, AutoGCN, and many more [52]. It's worth noting that many of these models are not mutually exclusive and can be combined to create more powerful models.

The background chapter has introduced a lot of concepts and techniques that are relevant to our work. We have discussed the importance of data preprocessing and feature engineering, explored various machine learning algorithms, and looked at some common neural network architectures. Although not exhaustive, this overview should provide a solid foundation for the methods, design and implementation that will be discussed in the next chapters.

3 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.

3.1 Hardware and software architecture

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

3.1.1 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 an 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 [46]. This means that the operating system is built by the Nix package manager, using a declarative configuration language. It 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.

Unfortunately, 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. Small bash and python scripts have been run on this laptop, as well as tests of the different developed programs, but all the main experiments have been run on the server.

In that context, we were provided 2 development servers towards the end of the thesis, around August 2023. The hardware 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

On this server, we have been given access to two docker instances running `Ubuntu 20.04.6 LTS`. The first instance is called **Deathstar** and the second one is called **Drogon**. For this Masterarbeit, I have mostly relied on Drogon for the final experiments, although Deathstar has been used for some preliminary experiments.

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.

3.1.2 Software, languages and tools

In Computer Sciences, it doesn'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.

Throughout 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 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.

3.1.2.1 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.

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 [43]. 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 developing on NixOS, the integration of Nix with the operating system is very good, and can be easily setup.

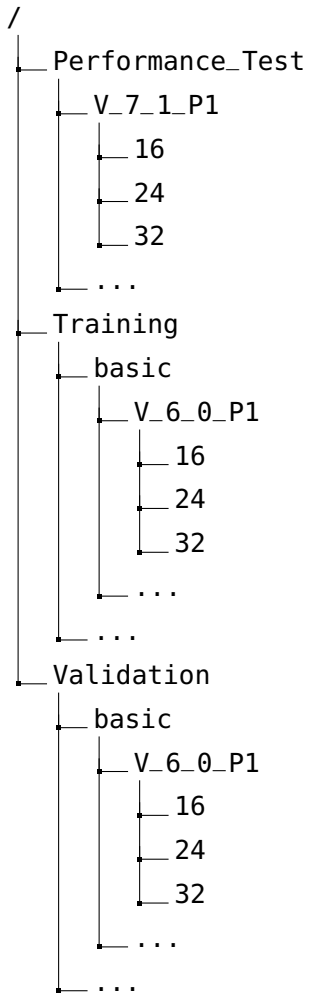
Nix is however really straightforward to install on any other distribution through the use of a single script available online. It can be installed in as little as one command.

3.2 OpenSSH memory dumps dataset

SmartKex has contributed to the research community by generating a comprehensive annotated dataset of OpenSSH heap memory dumps [9]. The dataset is publicly available on Zenodo [1].

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 3.1: 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.

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

Here is an example of content of a RAW memory dump file, displayed using *vim* and *xxd* 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 3.1: 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 3.2: 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 an 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 3.3: 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 used for key prediction.

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.

3.2.1 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 [6].
- **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» [5].

In the scripts and programs that have been developed 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.

3.2.2 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 3.4: 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 3.5: 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 3.6: 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`

3.2.3 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.

3.2.4 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 3.7: Count all dataset files

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

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

Listing 3.8: 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 3.9: 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 3.10: 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.

These 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 terms 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 2.1.1.2), 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.

3.2.5 Dataset validation

The dataset is merely 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 JSON annotation files are not valid JSON files, and cannot be loaded as such. Some 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 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 length) 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 3.11: Missing keys in JSON annotation file *Training/basic/V_6_0_P1/16/24375-1644243522.json*

3.2.5.1 Automatic annotation validation

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

Files are regrouped 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

```
1: Procedure ValidateJson(json_data)
2:     Initialize errors = Dictionnary{}                    ▷ Serve as collection for counted errors
3:     Initialize mandatory_json_keys = ['HEAP_START', 'SSH_STRUCT_ADDR', 'SESSION_-
   STATE_ADDR']
4:     Initialize key_names = {}
5:     Initialize incorrect_keys, missing_keys, incomplete_keys = 0
6:     for mandatory_json_key in mandatory_json_keys do ▷ Check if some expected json keys are
   missing
7:         if mandatory_json_key not in json_data or not correct hex address then
8:             errors[mandatory_json_key] = False
9:         else
10:            errors[mandatory_json_key] = True
11:        end if
12:    end for
13:    for json_key in json_data.keys() do          ▷ Get all the keys names, like A, B, C, D, E, F
14:        if json_key.startswith("KEY_") then
15:            key_name = GetLetterOfSSHKeyFromJSONKeyName(json_key)
16:            key_names.add(key_name)
17:        end if
18:    end for
19:    for key_letter in key_names do
20:        base_key = "KEY_" + key_letter
21:        PerformSSHKeyAnnotationValidationAndCompleteness(base_key, json_data) that
   counts incorrect_keys, missing_keys, incomplete_keys
22:    end for
23:    Store counters in errors
24:    return errors
25: 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 keys in general:

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

3.2.6 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** 3.2.1. 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.

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

Listing 3.12: 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 annotations, 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.

3.2.7 Exploring patterns in RAW heap dump files

Before diving into programming, we need to gain a better understanding of how to retrieve 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 3.2.

3.2.7.1 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 chosen to split the study of heap dump files in chunks or blocks of 8 bytes. The term *block* in code is always referring 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 3.13: 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 3.2.7.1:

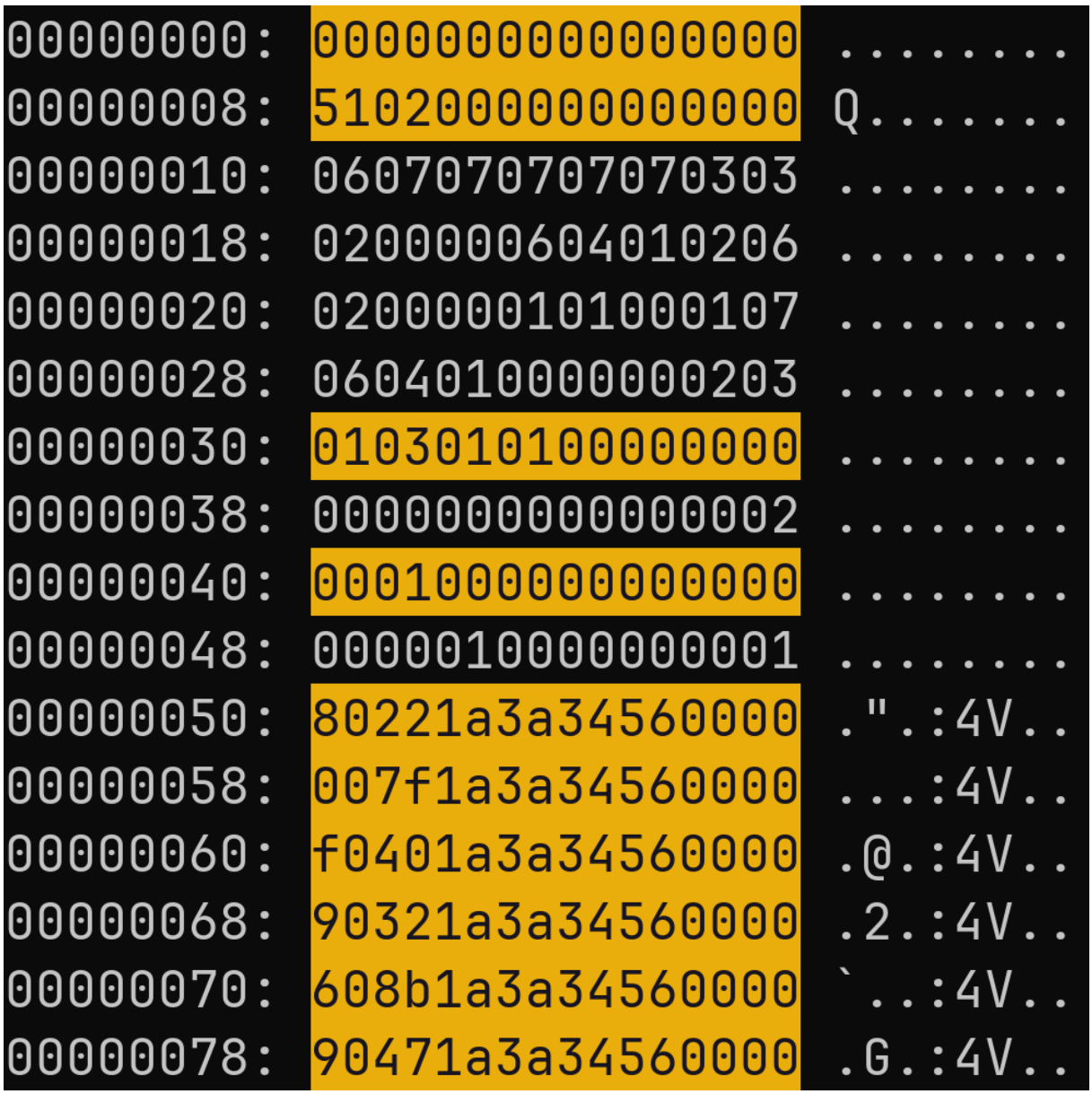


Figure 3.2: Binary RAW heap dump file loaded using `vim` and `xxd`, from */Training/Training/scp/V_-7_8_P1/16/1010-1644391327-heap.raw*, with highlight on rows with 12 hexadecimal digits followed by 4 zeros.

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 3.14: 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 3.15: 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 3.16: 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 pseudocode 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 pseudocode 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.

3.2.7.2 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“ [9]

```
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
```

This algorithm 3.2.7.2 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 [9].

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 pseudocode 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 relies 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 [9]. 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“ [9]

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 relies 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 algorithm 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 of 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 than two blocks is related to the minimum 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 3.2. 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 3.17: Python function to compute the Shannon entropy of a given block of 8 bytes

This function used NumPy 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 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 [9], but less that the 2% reduction obtained by the ML-based processing algorithm described in the paper [9]. 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.

3.2.8 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` 3.2.2. 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 malloc chunk has a hidden word of overhead holding size and status information» [5]. 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.

3.2.8.1 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 [5] [6]. Below are the key components of a chunk:

A chunk 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 [6] [7]:

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 chosen in the schematic 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 chunk 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 chunk of 8 byte blocks can be described by the diagram below [5]. Note that is representation is personal to better suit the needs of our forensic analysis. 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» [6]. 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 schematic representation, and does not change the actual data structure.

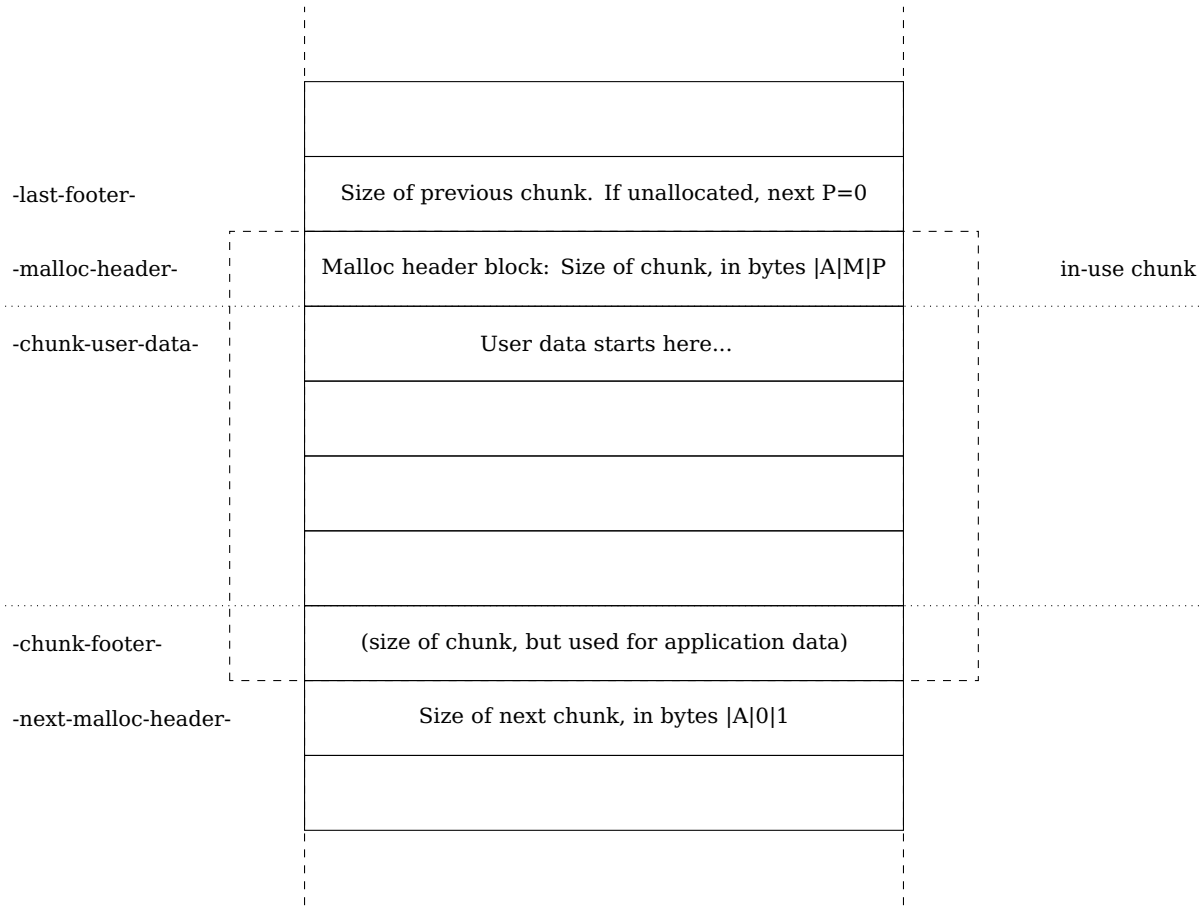


Figure 3.3: Diagram of an allocated chunk in GLIBC 2.28 [5].

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 [5] [6]. 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 3.18: 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 [5], is given below:

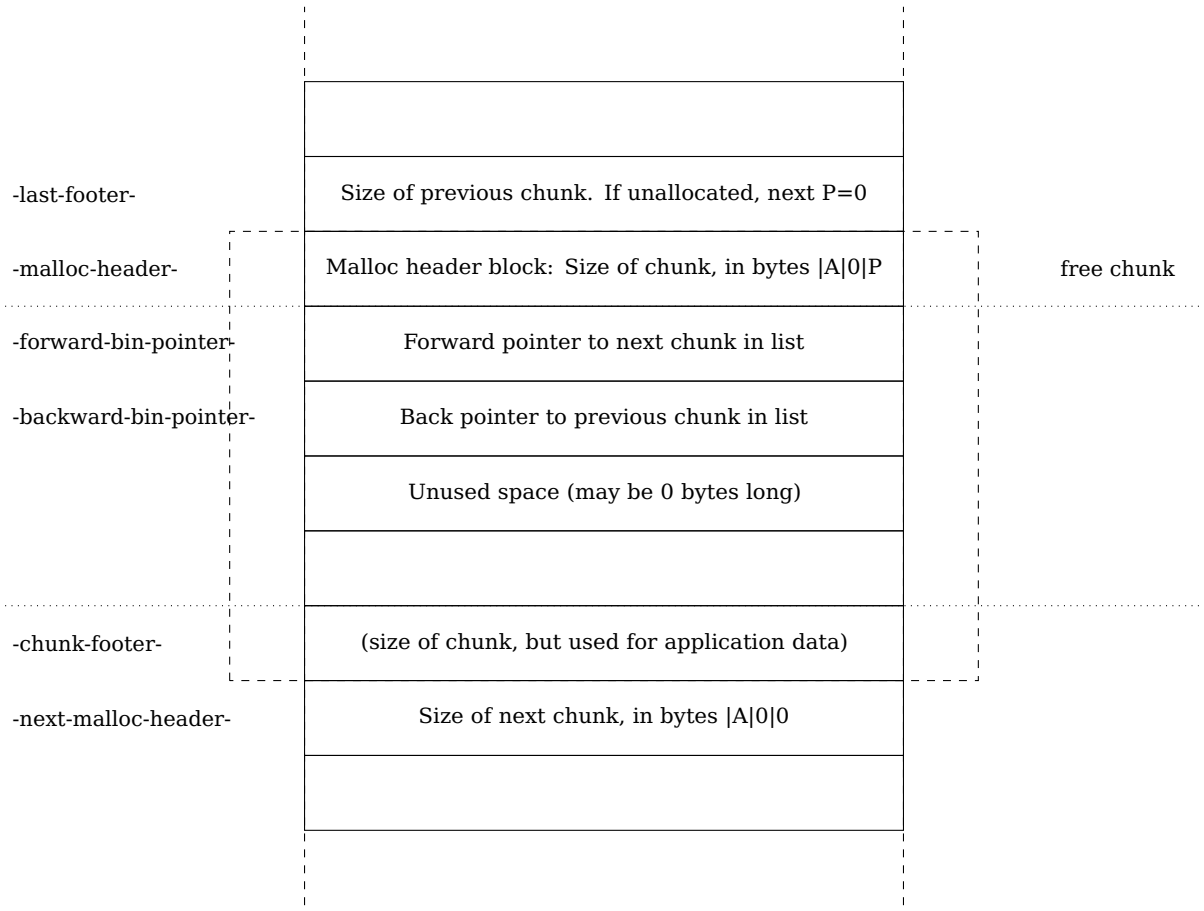


Figure 3.4: Diagram of a free chunk in GLIBC 2.28 [5].

3.2.8.2 Chunk chaining

The chunk chaining algorithm relies on the **chunk chaining assumption** 3.2.1. 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. It 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 3.19: 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 3.20: Vim command to find potential malloc headers

This gives something like the following:

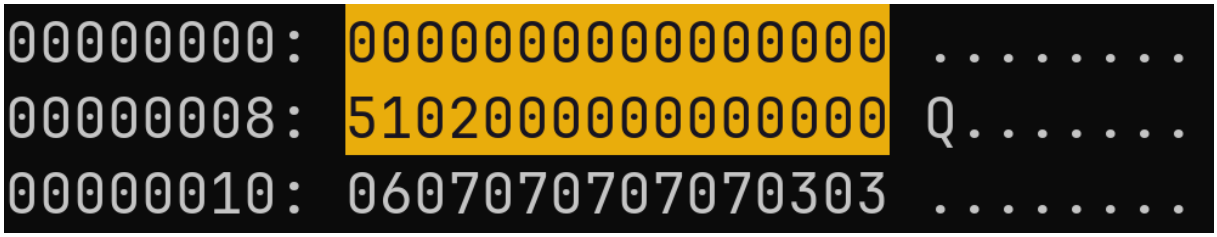


Figure 3.5: Attempt at malloc header detection in *Training/basic/V_7_8_P1/16/5070-1643978841-heap.raw*, at heap start.

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 3.6: 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 3.2.8.2
    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 behind 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. Then we 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 3.2.6.

3.2.8.3 Chunk chaining example

The program `chunk_algorithms.py` has been developed 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 3.21: 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.

3.2.8.4 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.

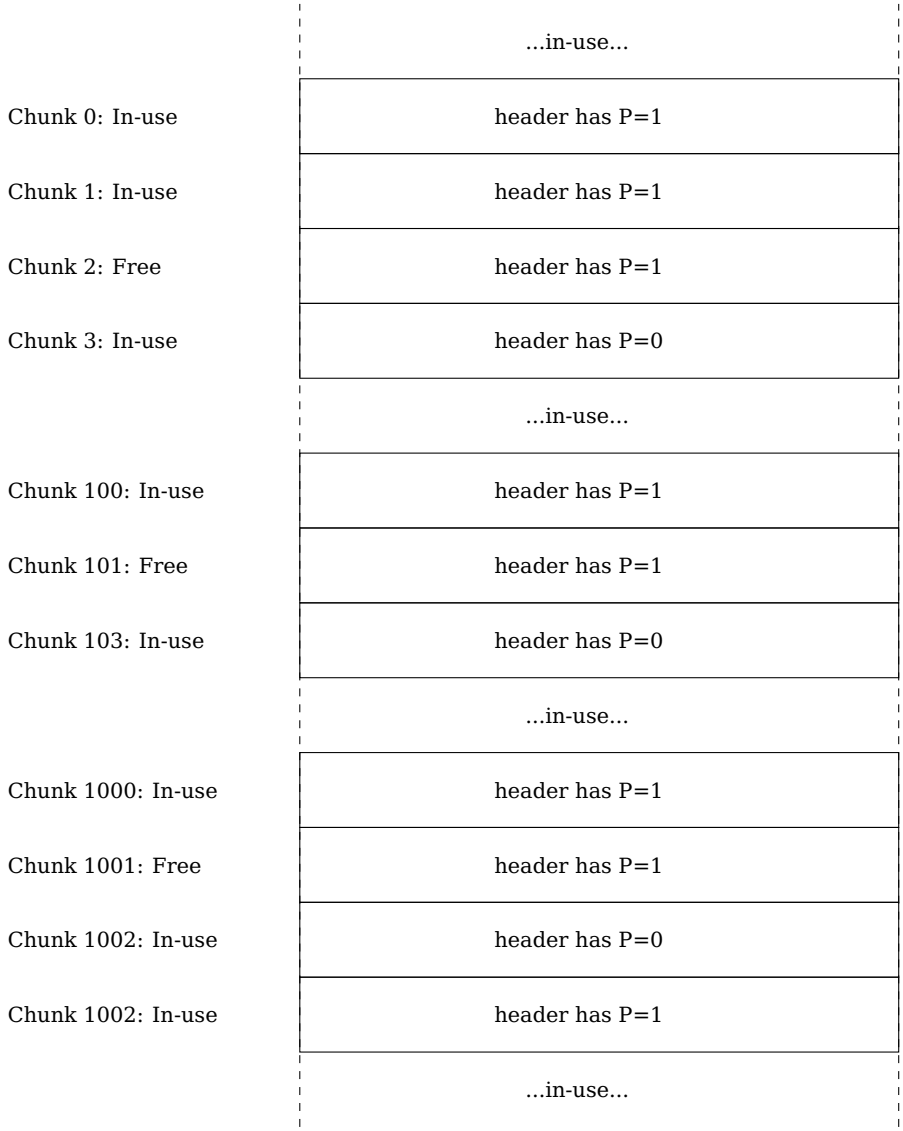


Figure 3.7: Heap dump showing a mix of free and in-use chunks. Note: each chunk immediately after a free chunk has a P flag set to 0. Each rectangle represents a chunk.

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 3.2.8.2
            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**
                    $Chunk.correct_{f}ooter \leftarrow True$
                else
                    $Chunk.correct_{f}ooter \leftarrow False$
                end if
            else
                $Chunk.correct_{f}ooter \leftarrow False$
            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 3.2.8.2. 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.

3.2.8.5 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 3.22: 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 looks 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.

3.2.9 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 RAW heap dump files, 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 developed 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 use, with correct footer, and annotated: 7668
Total number of chunks in use, 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 3.23: 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 value of 8, 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.

3.2.9.1 From a block-based to a chunk-based approach

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

Algorithm 11 Annotate Chunk Algorithm

```
1: 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
2:     Note: Annotations should be done after free chunk detection.
3:     Procedure AssertChunkUsedThenAnnotate(chunk, annotation)
4:         Assert chunk.is_in_use                ▷ Make sure we don't annotate free chunks
5:         chunk.annotations.append(annotation)
6:     end Procedure
7:     if chunk.address ∈ keys_addresses then
8:         AssertChunkUsedThenAnnotate(chunk, ChunkAnnotation.ChunkContainsKey)
9:     else if chunk.address = ssh_struct_addr then
10:        AssertChunkUsedThenAnnotate(chunk, ChunkAnnotation.ChunkContainsSSHStruct)
11:    else if chunk.address = session_state_addr then
12:        AssertChunkUsedThenAnnotate(chunk, ChunkAnnotation.ChunkContainsSessionState)
13:    end if
14: 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.

3.2.9.2 Using chunk footer for filtering is not possible

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.

3.2.9.3 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 allows to reduce the number of chunks from 37682063 to 37346373, which is a small reduction of 0.89%. It is not a huge reduction of chunks, but is a much more significant reduction of the number of blocks since 27.2% of the blocks are in free chunks.

We can also filter the chunks whose size is not 32, 48 or 64 bytes. This is based on the observation that the key chunks are of those sizes. Such a filtering technique allows to reduce the number of chunks significantly, since there is a cumulated $109346 + 34366 + 13434 = 157146$ chunks of those size, which represents, compare to the original number of chunks, a diminution of 99.6%. It is indeed a huge reduction of the scale of the problem.

A last approach to chunk filtering for key prediction consists in measuring the entropy of the first bytes of a chunk. Since we have previously discovered that keys are always located at the beginning of a chunk, and since the keys are composed of random bytes, we can expect the entropy of the first bytes of a chunk to be high.

The following graph illustrates this phenomenon:

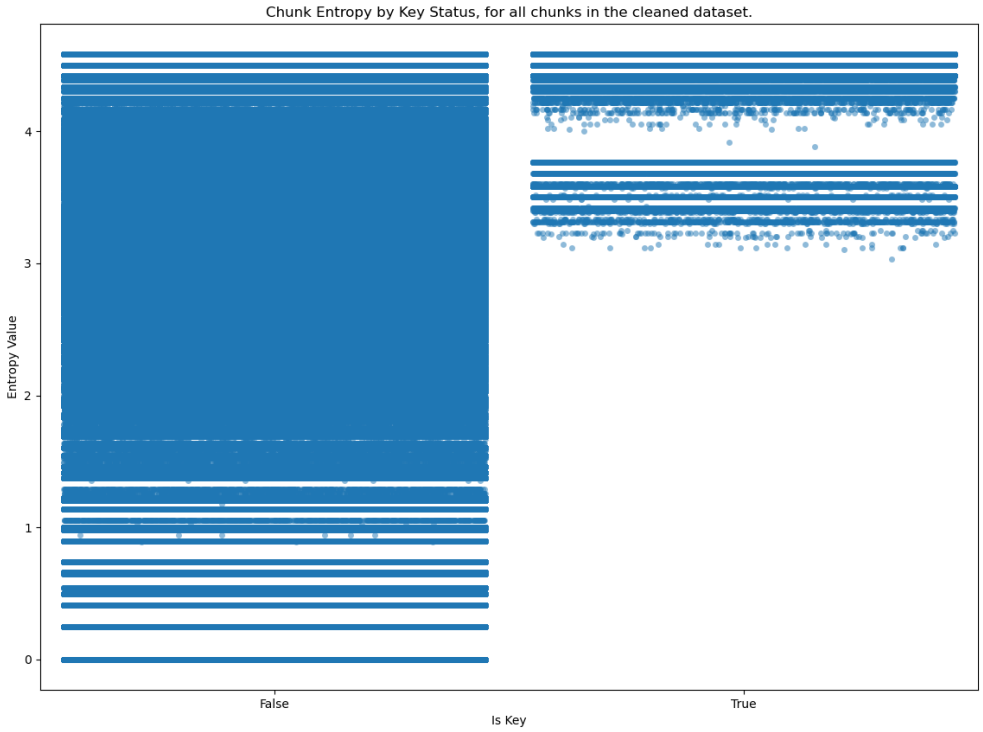


Figure 3.8: Visualization of the entropy distribution for all chunks of the *phdtrack_data_clean/* RAW heap dump dataset.

Using a script to perform some counting, we realize that the number of chunks whose entropy

is less than the minimum entropy of the chunks that contains a key (is key chunk) is 19690826, which represents 52.3% of the total number of chunks. It's not entirely clear why the key chunk entropy values seem spread across 2 strips of values, but is probably an effect of the different distributions of chunks across the input dataset, depending on the version, use can and number of chunks in the considered input files.

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 meaningful embeddings for the blocks we want to perform machine learning on. This is the goal of the next section.

3.3 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.

3.3.1 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, a cross validation pipeline, several balancing strategies and of course, a memory graph representation with a 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, 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, parsing algorithms 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.

Such an 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. It 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 represents 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 just a few hours.

3.3.2 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 Knowledge Graph ontologies. In the case of RDF, this could be equivalent to a **rdf:Bag**, which is an unordered container [8] (see 2.3.2.4). 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 inferred characteristics.

- **PN:** Pointer Node. This is a node whose bytes have been identified as a pointer.
- **CHN:** Chunk Header Node. This is a node whose bytes have been identified as a memory (malloc) chunk header. In the graph, this node is the root node of a memory structure managed by the C library responsible for memory management and allocation.
- **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.

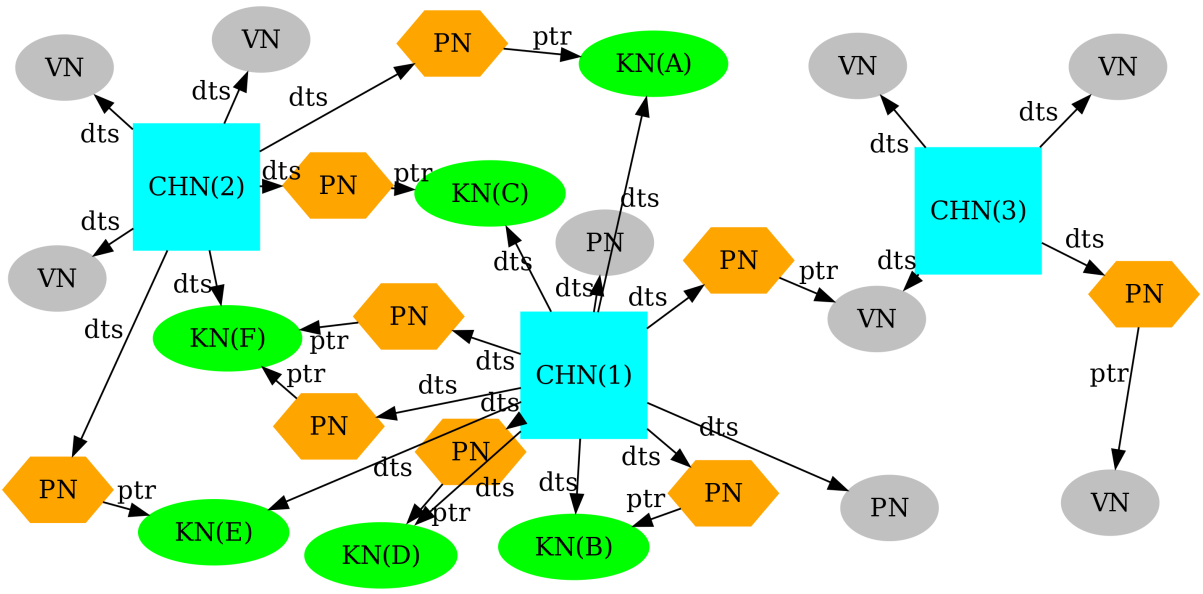


Figure 3.9: Visualization of a truncated memory graph generated from *Training/basic/V_7_1_-P1/24/17016-1643962152-heap.raw*. The addresses are not displayed for improved readability. Version with addresses here .2.

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 gray. 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 3.24: Command used to generate the memory graph visualization of *Training/basic/V_7_1_P1/24/17016-1643962152-heap.raw*

3.4 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.

3.4.1 Initialization and data checking

The graph construction process begins with the initialization phase. In this phase, the first step is graph initialization, which involves loading a given heap dump file and its associated annotation file. Once the files are loaded, several checks are performed on the annotation file to ensure its validity. These checks include verifying that all annotations are present and formatted correctly, as well as ensuring that the annotation file is neither empty nor contains errors.

3.4.1.1 Graph Construction steps

The second major step in the process is graph building. This involves constructing the graph from heap dump byte blocks. The first part of this step is the data structure detection, where blocks are parsed from start to finish. The parsing process leaps over blocks by using chunk sizes that are stored in chained chunk headers. Each chunk is then verified for its size, alignment, and the presence of a potential footer. Following this, the pointer detection step is carried out. In this phase, potential pointers are identified using the previously introduced pointer detection algorithm and are added to the graph.

An optional step that can be performed is the chunk pointer reduction. This step removes any blocks that are not Chunk Header Nodes, effectively transforming the graph from a block-based graph to a chunk-based graph. While this step is not mandatory, it can be useful for reducing the scale of the problem. This approach will be extensively used in the machine learning section, as it has been shown that the key block prediction problem is equivalent to a key chunk prediction problem.

3.4.1.2 Graph Annotation

The third major step in the process is graph annotation. In this phase, Value Nodes in the graph are replaced by Key Nodes, utilizing the annotations provided in the JSON annotation file. Additional

annotations, such as SSH_STRUCT, can also be added at this stage. Following the completion of the annotation step, it becomes possible to export the graph for various purposes. The graph can be exported to file formats like `.dot` or `.gv`, which are suitable for visualization or other analytical tasks.

At this step, the graph is looking similar to the example shown before 3.3.2. Using the pointer reduction to a chunk-only memory graph, we can obtain something like the following:

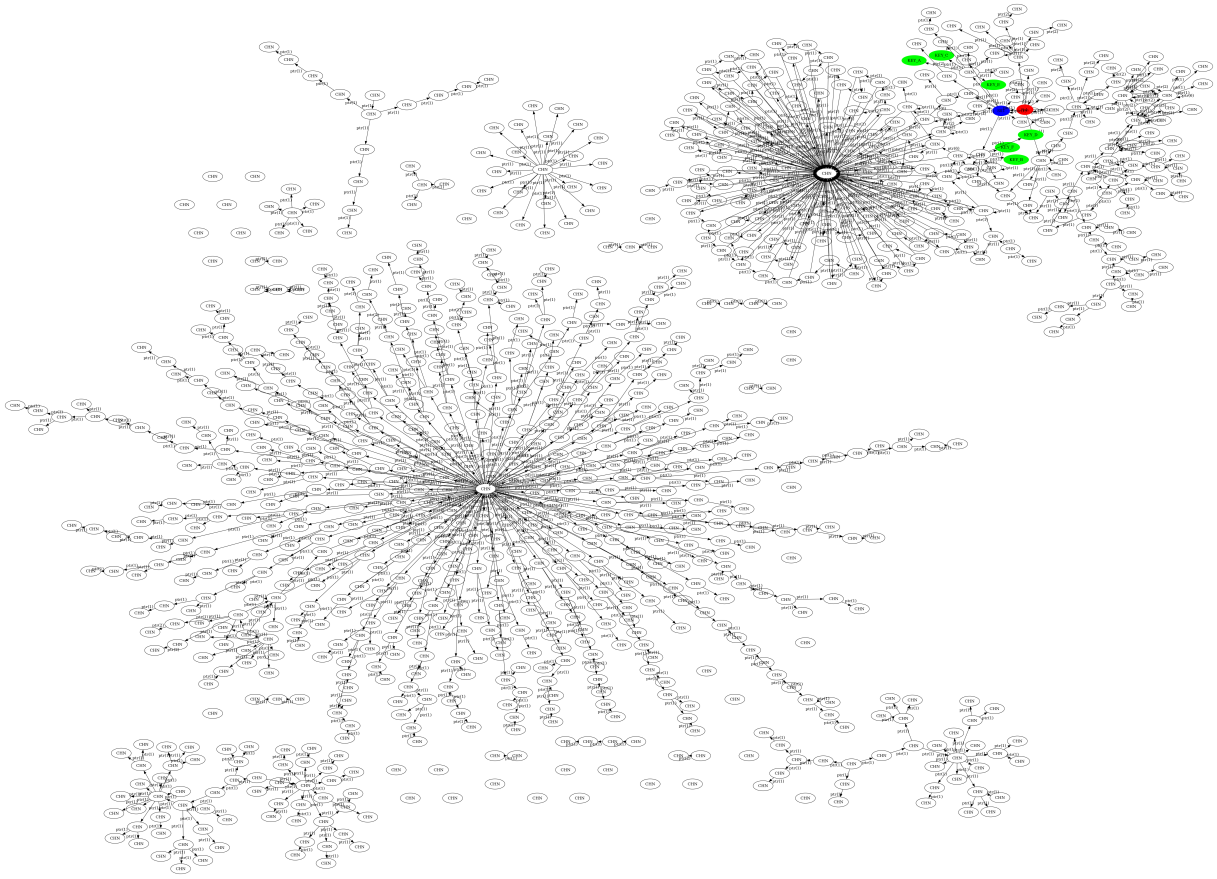


Figure 3.10: Visualization of the chunk memory graph, with only Chunk Header Nodes representing chunks, generated from *Training/scp/V_7_8_P1/16/ 585-1644391327-heap.raw*.

Note how we can identify some data structures formed by pointer-connected chunks. This is a very interesting property of the memory graph representation, since it allows identifying data structures and their members based only on the shape of connections. This is a very important property that will be used later for feature engineering and embeddings.

3.4.1.3 Custom Graph-Based Embeddings

The fourth step in the workflow involves generating embeddings from the graph. Multiple types of embeddings can be generated, each serving a unique purpose and offering different insights into the graph structure. One such embedding is the semantic embedding. This is a general approach that enriches each node with information related to its graph structure vicinity. It captures the essence of the node's position and relationships within the graph, making it useful for various machine learning and analytics tasks.

In addition to semantic embeddings, other types of embeddings can also be generated. For

instance, statistical embeddings focus on capturing the statistical properties of the graph. Another interesting type of embedding is the random walk embedding and related version called Node2Vec. This method leverages the random walk algorithm to generate embeddings, capturing the local and global structure of the graph by simulating random paths through it.

Each of these embeddings offers a unique lens through which to analyze and interpret the graph, and the choice of embedding can be tailored to the specific requirements of the task at hand. It's also possible to add additional features like entropy of the chunk start bytes and filtering information.

3.4.1.4 Exporting the Graph

The fifth step in the process involves exporting the graph. The graph is exported to a `.gv` DOT file format. Custom embeddings are integrated into the graph by utilizing the comment fields in a slightly modified stringified JSON format. This approach allows for easy reading of the embeddings associated with each node while maintaining the DOT file as a valid format that can be used with tools and libraries supporting the DOT graph formal.

The DOT (Directed Orthogonal Text) format is a plain text graph description language that is widely used for representing structured information. An example of a memory graph in DOT format without embeddings is provided for reference. In this example, nodes and edges are represented along with their attributes such as color, shape, and labels.

When embeddings are added to the graph, additional comment fields are included in the DOT file nodes. These comment fields contain a JSON string that holds the embedding information for each node. Moreover, the graph starts with a pseudo JSON comment field that contains a serialized JSON object specifying the embedding type and feature names.

Here is an example of a comment field in a node:

```
1  comment="[0,94918015119368,1,108,108,108,108,108,108,108,1,67,82,139,175,204,...
2  0,0,2.355388542207534]"
```

Listing 3.25: A comment field example for a node with embedding. Output is cropped.

Below is an example of a memory graph comment field containing a JSON serialized object with embedding type and feature names.

```
1  comment="{ 'embedding-type': 'chunk-semantic-embedding', 'embedding-fields': ['
       block_position_in_chunk','chn_addr','chns_ancestor_1',...'chns_children_8','
       chunk_byte_size','chunk_number_in_heap','chunk_ptrs','chunk_vns','ptrs_ancestor_1
       ','ptrs_ancestor_2',...,'ptrs_children_8','entropy'] }"
```

Listing 3.26: A memgraph comment field example containing JSON serialized object with embedding type and feature names. Output is cropped.

The inclusion of these comment fields serves multiple purposes. First, it allows for the storage of embeddings along with additional information, making the graph more informative. Second, this format can be easily integrated into Python machine learning pipelines, facilitating the use of the graph in various machine learning tasks. Lastly, the DOT format serves as a standard format for graph representation, making it a versatile choice for both visualization and computational tasks.

3.5 A wide range of features and embeddings

In the following, we will explore the features and embeddings developed and used in this thesis. We will start by the features and embeddings based on the memory graph characteristics, then we will explore the graph-agnostic embeddings, and finally, we will discuss the machine learning models used for the binary classification task.

3.5.1 Embeddings based on custom features

While doing the construction of the graph, we can add some custom features to the nodes. Those features have been developed to embed the unique characteristics of each node of the graph, depending on its type, parent chunk characteristics as well as its vicinity in the memory graph. Those features have been regrouped in several custom embeddings that will be discussed in the following.

3.5.1.1 Remark on the collaborative work

This section focuses on the embeddings developed during a Masterarbeit project around OpenSSH heap dump analysis. Clément Lahoche and the author of the present report have done a collaborative effort on the matter. Since Clément has focused on the embeddings, the following section will not discuss in too many details the embeddings that are already described and analyzed in details by Clément's work. The reader is invited to read his Masterarbeit report [2] for more information. This section includes some elements that are clearly identified as coming from Clément's work.

As a notable difference, the Node2Vec embedding, which is a graph-agnostic embedding, will also be discussed in the following, which is not the case in Clément's thesis. This is because the present report focuses on the machine learning part, and especially on graph representation learning.

3.5.1.2 Semantic graph embedding

The focus of this stage is on semantic embedding, a technique that transforms the graph into a low-dimensional vector space. Each vector encapsulates the local neighborhood of a graph chunk, enabling the application of advanced machine learning methods. The embedding process is intricate, considering both direct and indirect connections to and from each chunk. It starts by counting the number of pointers and chunks directly linked to a specific chunk, and then extends this by recursively exploring deeper layers of connections. A parallel reverse analysis is also conducted to capture child nodes. The outcome is a compact vector that richly represents the chunk's contextual relationships within the graph.

The following algorithm describes the process of generating the semantic embedding for a given chunk:

Algorithm 12 Generate Ancestor/Children Embedding.

```
function GenerateNeighborsCHN(chunk_node, dir)
    ancestor_nodes ← an empty set
    children ← graph.neighbors_directed(chunk_node, OUT)        ▷ Get members of the chunk
    for child in children do
        ancestor_nodes.insert(child)
    end for
    result ← an empty list
    current_nodes ← an empty set
    for _ in 0 to DEPTH do
        current_nodes ← ancestor_nodes                ▷ switch ancestor nodes and current nodes
        ancestor_nodes ← an empty set
        nb_chn ← 0
        nb_ptr ← 0
        for current_node in current_nodes do
            if node is ChunkHeaderNode then              ▷ Update number of chunks and pointers
                nb_chn ← nb_dtn + 1
            else if node is PointerNode then
                nb_ptr ← nb_ptr + 1
            end if                                        ▷ Get neighbors of the current node
            for neighbor in graph.neighbors_directed(current_node, dir) do
                ancestor_nodes.insert(neighbor)          ▷ Add neighbors to the next ancestor nodes
            end for
        end for
        result.append(nb_chn)                                ▷ Add number of data structures
        result.append(nb_ptr)                                      ▷ Add number of pointers
    end for
    return result
end function
```

Note that this algorithm is taken from Lahoche, from his Masterarbeit report [2]. It has been developed and implemented as a collaborative effort on this project.

The embedding algorithm is applied to each chunk in the graph, exploring up to a predefined depth, generally 8, which is a hyperparameter of this embedding. This results in a 32-unit embedding, broken down into 8 units each for ancestor pointers, ancestor chunks, child pointers, and child chunks. Basic chunk attributes like block position, byte size, and number of pointers and value nodes are also included, bringing the total embedding size to 37 units. Despite its comprehensiveness, the embedding has limitations, such as the potential for noise from value nodes and the complexity of capturing intricate relationships.

A way that have been used extensively, to both reduce the number of nodes and to improve the quality of the embeddings, is to reduce the graph to a chunk-only graph. This is done by removing all the nodes that are not chunk header nodes. It is a very interesting approach, since it allows to reduce the scale of the problem by a factor of 10, and also allows focusing on the data structures, which are the most interesting nodes to embed. This approach will be used in the machine learning section, since we have shifted the focus from block to chunk prediction.

3.5.1.3 Semantic features from essential chunks attributes

Every chunk in the heap dump comes with fundamental attributes that provide insights on its structure and content. These attributes are not limited to the primary chunk nodes but are also inherited by value and pointer nodes, which are subcomponents of a chunk. The key attributes include the block's position within the chunk, the chunk's byte size, the total number of pointers and value nodes in the chunk, and the chunk's index in the heap. These details collectively offer a thorough understanding of each chunk's makeup and its relative position in the heap.

3.5.1.4 Statistical Embedding

Statistical embeddings serve as a powerful tool for reducing high-dimensional data while preserving essential patterns and probabilistic relationships. One key technique employed is the use of n-gram values, specifically focusing on bit combinations to manage dimensionality. This approach aligns with the primary goal of identifying SSH keys, which inherently display a wide range of frequencies. Various n-gram sizes are utilized, including 1-gram, 2-gram, 3-gram, up to 8-gram, with the latter contributing significantly to capturing broader contextual patterns.

In addition to n-grams, other statistical metrics like mean, standard deviation, MAD, skewness, kurtosis, and Shannon entropy are incorporated. These metrics offer a multi-faceted view of the data, aiding in the identification of SSH keys. However, chunks with a standard deviation of zero are excluded from the analysis, as they are unlikely to contribute to the identification of random patterns like SSH keys. These skipped values are replaced with `NaN` values in embedding comment of nodes, that needs to be handled by the machine learning pipeline. It has been chosen to replace those values with zeros.

Finally, the statistical embedding vector for each chunk is constructed by combining n-gram values and these additional statistical metrics. The vector also includes basic chunk information, resulting in a comprehensive vector that encapsulates the chunk's characteristics.

3.5.1.5 Start-bytes Embedding

In addition to the aforementioned embeddings, a simpler approach was implemented to serve as a baseline for comparison. This method focuses solely on the initial bytes of each chunk for vectorization. The sample vector is initialized with basic chunk information and then populated with the first bytes of the chunk, up to a predefined limit. If the chunk has fewer starting bytes than the predefined limit, zeros are added to fill the remaining positions. This straightforward approach provides a straightforward embedding, suitable for comparative evaluations with more intricate embeddings.

3.5.2 Embedding transformations depending on the model

When it comes to feeding embeddings into machine learning models, the shape and size of the embeddings need to be tailored to fit the model's requirements. For classic machine learning algorithms like Random Forest, the requirement is that the embedding matrices must be of a fixed, predefined size. This presents a unique challenge when working with graph embeddings, as graphs usually have a variable number of nodes. To address this, padding is added to the embedding matrices to ensure they all match the size of the largest graph. In contrast, Graph

Convolutional Networks (GCN) offer more flexibility in this aspect. GCNs are capable of handling variable-size embedding matrices as long as the number of features is fixed. This eliminates the need for padding the matrices, which is advantageous as it simplifies the preprocessing steps.

3.5.2.1 Node filtering to feature

On the topic of node filtering in the context of chunk memory graphs, it's important to note that active rebalancing isn't performed, despite the number of positive nodes (key chunks) being substantially lower than the number of negative nodes (non-key chunks). The rationale behind this choice is to enable models to learn from complete graphs. This is particularly relevant for Graph Convolutional Networks, which are capable of handling graphs of variable sizes. The goal is for the models to be able to identify key chunks even in completely unlabeled memory graphs. The ability of GCNs to process variable-size graphs makes them especially suitable for this kind of task, as it allows the model to learn from the full structure of the graph without the need for compromising the integrity of the data through techniques like rebalancing.

3.5.3 Graph-agnostic Embeddings

Unlike our previous embeddings, which were developed manually to suit the intricacies of chunk graphs, there are also pre-existing, generalized graph embeddings. These graph-agnostic embeddings offer the benefit of being applicable to a wide range of graphs without requiring specific customization based on the characteristics of the underlying data.

3.5.3.1 RandomWalk

The RandomWalk algorithm offers a straightforward approach to graph embedding. It simulates random walks starting from each node in the graph and uses these walks to create vector representations of the nodes. One of the advantages of RandomWalk is its simplicity, both in terms of implementation and interpretation. The algorithm excels at capturing local structures within the graph, making it particularly effective for tasks such as community detection and link prediction. However, its focus on local characteristics means that it might not capture global properties of the graph as effectively.

3.5.3.2 Node2Vec

Node2Vec extends the capabilities of the RandomWalk algorithm by introducing additional parameters that allow for a more nuanced exploration of the graph. This makes Node2Vec more versatile than RandomWalk, enabling it to capture both local and global graph structures. Because of its flexibility, Node2Vec is well-suited for a range of applications including the specific task of providing an embedding for the node of memory graphs. While its versatility is a strong point, it comes at the cost of increased computational complexity due to the introduction of several hyperparameters.

Hyperparameters:

- **p:** The return parameter, which controls the likelihood of the walk returning to the node it just left.

- **q:** The in-out parameter, which differentiates between inward and outward nodes in the walk.
- **Length of Walk:** Determines the length of each random walk.
- **Number of Walks:** Specifies the number of walks to initiate from each node.

Note that the two last hyperparameters have a huge impact on the performance of the algorithm. These hyperparameters play a crucial role in shaping the behavior of the Node2Vec algorithm. Specifically, the return and in-out parameters help guide the random walks in a way that allows the algorithm to capture different types of structural information from the graph. The length and number of walks, meanwhile, impact the granularity and quality of the embeddings generated.

3.6 Machine Learning Binary Classification

Binary classification is a type of machine learning task where the model is trained to differentiate between two classes. In the context of key chunk prediction, binary classification serves to identify whether a given chunk is a "key chunk" or not. Successfully predicting key chunks is crucial as it leads to a 100% successful key retrieval rate. This is because, in our case, all keys are situated at the beginning of a chunk, and no chunk contains more than one key. Various machine learning models, ranging from classic approaches to more modern methods like Graph Convolutional Networks (GCNs), have been employed for this task.

3.6.1 Classic Models of Machine Learning

For baseline comparisons, we have experimented with classic machine learning models including Random Forest, Logistic Regression, and the SGD Classifier. These models serve as a well-studied and understood starting point for our classification problem, providing a frame of reference against which more complex models can be compared.

3.6.1.1 Random Forest

Random Forest is an ensemble learning method that operates by constructing multiple decision trees during training and outputs the class that is the mode of the classes of the individual trees for classification tasks. It is highly flexible and can handle a wide range of data types, making it a strong candidate for various use cases, including key chunk prediction.

Strong and Weak Points:

- **Strong:** The model is robust to overfitting and can handle high dimensional data well.
- **Weak:** Random Forest models can be computationally expensive and may require a long training time, especially for larger datasets.

As often with models, Random Forest has a bunch of hyperparameters:

Hyperparameters:

- **n_estimators:** Number of trees in the forest.
- **max_features:** The number of features to consider when looking for the best split.
- **max_depth:** The maximum depth of the tree.
- **min_samples_split:** The minimum number of samples required to split an internal node.
- **min_samples_leaf:** The minimum number of samples required to be at a leaf node.

We have used the implementation of Random Forest available in the Scikit-learn library [26].

3.6.1.2 Logistic Regression

Logistic Regression is a statistical model commonly used for binary classification tasks. It models the log-odds of the probability of the event occurring as a linear combination of the predictor variables. The model is particularly effective when the probability of the outcome (dependent variable) can be expressed as a logistic function of the predictor (independent variable).

Strong and Weak Points: Logistic Regression is straightforward to implement and understand, making it a good starting point for many classification problems. However, its simplicity is both a strength and a weakness; it might not perform well when the relationship between the variables is not log-linear or when the dataset has high dimensionality.

Hyperparameters:

- **C:** Inverse of regularization strength; smaller values specify stronger regularization.
- **solver:** Algorithm to use for optimization, such as 'liblinear' or 'saga'.
- **max_iter:** Maximum number of iterations for the solver to converge.

We have relied on the Logistic Regression implementation available in the Scikit-learn library for our experiments[26], using the default hyperparameters.

3.6.1.3 SGD Classifier

The Stochastic Gradient Descent (SGD) Classifier is a linear classifier optimized by stochastic gradient descent. It is especially useful for large-scale and sparse machine learning problems. The SGD Classifier can approximate other types of linear classifiers like Logistic Regression and Support Vector Machines.

Strong and Weak Points: SGD Classifier is computationally efficient, making it well-suited for large datasets. However, it requires careful tuning of its hyperparameters and might be sensitive to feature scaling.

Hyperparameters:

- **alpha:** Regularization term that discourages large coefficients to prevent overfitting.
- **loss:** Specifies the loss function to be used, such as 'hinge' for SVM or 'log' for logistic regression.

- **max_iter:** Maximum number of passes over the training data.
- **learning_rate:** The learning rate schedule, could be 'constant', 'optimal', 'invscaling', or 'adaptive'.

For the SGD Classifier as well, we used the Scikit-learn implementation[26].

3.6.2 Graph Convolutional Networks (GCN)

As GCNs have already been introduced in the background section, this part will primarily focus on our specific implementation and the variants of GCN models employed for the task of key chunk prediction.

GCNs are a specialized form of neural networks designed to operate directly on graphs. One of their main characteristics is their ability to capture the graph's structural information. They accomplish this by using edge connectivity information, either from an adjacency matrix or an edge list, as part of their input. This makes them particularly effective for tasks involving irregular data structures like the memory graphs discussed previously.

We used the PyTorch Geometric library for the development of our GCN models. This library offers a robust set of tools and abstractions, making it easier to construct custom graph-based neural networks [3].

GCNs excel at capturing the topological features of graphs, making them a strong candidate for our task of key chunk prediction. However, they can be computationally intensive, especially for large graphs, and also require careful tuning of hyperparameters for optimal performance.

3.6.2.1 Very Simple GCN

The Very Simple GCN model is a minimalist approach, capturing the essential features of a Graph Convolutional Network. This model consists of just one graph convolution layer followed by a single fully connected layer. The convolution layer takes the input features and transforms them into a 16-dimensional space.

After the graph convolution operation, a ReLU (Rectified Linear Unit) activation function is applied to the output. ReLU is defined as $f(x) = \max(0, x)$, replacing all negative values in the tensor with zeros. In the context of GCNs, ReLU is commonly used to introduce non-linearity into the model. It enables the network to learn from the error and make adjustments, making it more capable of handling complex, non-linear relationships in the graph data.

Finally, the output from the ReLU activation is passed through a fully connected layer to produce the final output for classification. Due to its simplicity, this model is computationally efficient but may not capture complex graph structures effectively.

3.6.2.2 Simple GCN Models

The Simple GCN model, or `LessSimplifiedGNN`, is a step-up in complexity from the very simplified version. It incorporates two graph convolutional layers, doubling the depth of the network. The first convolution transforms the input features to a 12-dimensional space, and the second one

further transforms these 12-dimensional vectors into 24 dimensions. Two fully connected layers follow, ultimately producing the final output. This model is capable of capturing more complex features in the graph but is computationally more demanding than the simpler model.

3.6.2.3 First GCN Model

The First GCN model is the first version that was actually implemented and tested for the task. It's a more complex architecture optimized for higher performance. It consists of two graph convolution layers that transform the input features first into a 16-dimensional and then into a 32-dimensional space. Following these, the network contains three fully connected layers. These layers are intended to capture more intricate patterns in the data. Failed attempts to scale the model computation speed by delegating the fully connected layers to the GPU were made, but ultimately, the model was too memory intensive to be trained on the GPU. This model is designed for capturing more nuanced relationships in the graph but at the cost of higher computational requirements.

3.6.3 The Impact of Complexity

The complexity of a model is a double-edged sword when it comes to performance metrics like precision and recall. On one hand, more complex models with additional layers or more neurons can capture intricate patterns in the graph data, potentially improving precision by accurately identifying key chunks. On the other hand, the complexity often leads to overfitting, especially when the training dataset is small or lacks diversity. Overfitting can adversely affect recall, as the model might become too specialized in recognizing training graphs and fail to generalize well to unseen data.

In the case of Graph Convolutional Networks (GCNs), the non-linear transformations and multiple layers can allow the network to learn highly specialized features, which is excellent for achieving high precision. However, these can be detrimental to recall if the model becomes so tailored to the training data that it fails to identify key chunks in new, unseen graphs.

The impact of the model's complexity also varies with the number of input graphs. For datasets with few graphs, a simpler model is often more appropriate to prevent overfitting. In contrast, when numerous diverse graphs are available for training, a more complex model can be employed to capture the rich set of features inherent in the data, thereby potentially improving both precision and recall.

Therefore, the choice of model complexity should be carefully considered, weighing its impact on performance metrics and the size and diversity of the available training data. This is why more complex GCN models have also been implemented and tested.

3.6.4 Understanding Dropout in GCNs

Dropout is a regularization technique used in neural networks, including Graph Convolutional Networks (GCNs) and Convolutional Neural Networks (CNNs), to prevent overfitting. The dropout layer randomly sets a fraction of the input units to 0 during training, which helps to make the model more robust and improves generalization. In the ML pipelines developed in Python, a dropout rate of 0.5 is used, meaning approximately 50% of the neurons will be turned off during

each forward pass. This section will discuss the effect of dropout on the model's performance and the intuition behind it.

3.6.4.1 Effect on Binary Classification

In a binary classification problem, dropout can have several effects:

- **Reduced Overfitting**: By randomly deactivating neurons, the model is less likely to rely on any single feature, making it generalize better to unseen data. This is expected to reduce overfitting and thus improve the model's performance, especially the recall of the positive class.
- **Increased Robustness**: Dropout can make the model more robust to noise in the training data.
- **Variable Performance**: While dropout can improve generalization, it may also lead to increased variance in the model's predictions, especially if the dropout rate is too high.

In the code of some GCN models implemented, dropout is applied after the activation functions of the GCN layers and the first fully connected layer:

```
x = F.relu(x)
x = self.dropout(x)
```

This is a common practice as it allows the model to learn more robust features. However, the dropout rate and its placement in the architecture are yet other hyperparameters that may need to be tuned for optimal performance. Since we already have a large number of hyperparameters, we will not tune the dropout rate in this thesis.

3.6.4.2 Batch Normalization

The code also includes Batch Normalization layers, denoted by `self.bn1` and `self.bn2`. These layers normalize the features to have zero mean and unit variance, which can accelerate training and provide some regularization effect, complementing the dropout layers.

3.6.5 More Complex GCN Models

In an effort to enhance the performance of our initial GCN models, we explored more complex architectures. The motivation behind increasing the complexity was to evaluate whether additional layers or techniques could lead to improved performance metrics such as precision and recall. These advanced models have been tested against the simpler GCN models to determine their effectiveness.

3.6.5.1 GCN with Dropout and Batch Normalization

Building upon the First GCN model, we incorporated dropout and batch normalization layers. Dropout is employed to mitigate the issue of overfitting, especially relevant for complex models

trained on limited data. With a dropout rate of 0.5 after each ReLU activation, we were able to regulate the model's complexity during training.

Batch normalization, on the other hand, aims to accelerate training and stabilize the learning process. By normalizing the output features of each layer, batch normalization helps in alleviating internal covariance shift, making the model training more resilient to the choice of hyperparameters.

This version of the model aligns closely with the First GCN model, but the addition of dropout and batch normalization layers offer greater robustness, especially when training on imbalanced or smaller datasets.

3.6.5.2 Advanced GCN

The Advanced GCN model represents the most intricate architecture we have experimented with. This model introduces several additional components compared to the simpler models.

The Advanced GCN consists of three Graph Convolution layers, followed by Batch Normalization layers. The initial Graph Convolution layer transforms the input features into a 32-dimensional space, which is then normalized using `BatchNorm1d`. The second and third Graph Convolution layers further increase the dimensions to 64 and 128, respectively, also followed by batch normalization steps. These layers aim to capture more complex features from the graph structure.

ReLU (Rectified Linear Unit) activation functions are applied after each batch normalization, introducing non-linearity to the model and helping to capture intricate relationships in the data. Additionally, dropout layers with a rate of 0.5 are placed after each ReLU activation to mitigate the risk of overfitting.

After the Graph Convolution layers, the architecture includes a series of fully connected layers that transform the 128-dimensional feature vector into a 256-dimensional space, which is further compressed into 128 and then 64 dimensions. Finally, the model outputs a vector of dimensions corresponding to the number of classes. Each of these fully connected layers is followed by ReLU activations, except for the final output layer.

- **Graph Conv Layers**: Three layers with dimensions 32, 64, and 128
- **Batch Normalization**: Applied after each Graph Conv layer
- **Fully Connected Layers**: Layers with dimensions 256, 128, and 64
- **Dropout**: Applied after each ReLU activation with a rate of 0.5

In summary, the model is engineered to capture more complex relationships in the data, at the cost of increased computational requirements and a higher risk of overfitting, especially when trained on small or less diverse datasets. The dropout and batch normalization layers are integrated to combat overfitting to some extent.

The current 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 developed programs, experimentations conducted and subsequent results.

4 Results

The following section describes the experimental setup, the used and generated datasets as well as parameters to conclude with the experimental results achieved.

4.1 Developed programs

Many programs have been developed for the need of the present thesis. Early data exploration scripts have paved the way towards efficient highly parallel programs in both Rust and Python for data analysis, graph and embedding generation of ML and DL tested on different models and contexts. All the necessary concepts and methods have been introduced previously, so it is now time to present those main program in details.

4.1.1 Mem2Graph

The present report has already presented the *Mem2Graph* program throughout the heap dump memory parsing algorithm, graph construction and embeddings. This program has been developed in Rust and is an active collaboration between the author and Clément Lahoche, another PhDTrack student at Passau. The program is available on GitHub at `https://github.com/passau-masterarbeit-2023/mem2graph`.

The program is composed of several layers that build on top of each other. The first layer is dedicated into loading a RAW heap dump file with its annotation JSON file, performs some checks and prepare the data for further analysis. The next one performs the graph construction following the algorithms introduced in the Methods section. Another layer performs some annotations of the nodes. The final layer is more versatile and dependent on the input program parameters. For the need of this report, several pipelines of memgraph with and without embeddings have been added, namely `graph` and `graph-with-embedding-comments`. The first pipeline doesn't perform any embedding and export the memory graph to a text file following the DOT format [4]. The second pipeline performs the same operations, but also exports the memory graph with the embeddings as comments in the DOT file.

Since the prediction effort is focused on memory chunks, this embedding is generally called with the `-no-value-node` parameter, which transform the memory graph from a graph of blocks of 8 bytes, into a memory graph of memory chunks, connected by their pointers inside their respective user data space. Several chunk node embeddings have been implemented, and the user can choose which one to use. The chunk node embeddings are the following: `chunk-semantic-embedding`, `chunk-statistic-embedding`, and `chunk-start-bytes-embedding`.

4.1.2 Machine Learning Pipelines Runner

When working with machine learning, Python is a dominant language, benefiting from a rich ecosystem of libraries and frameworks.

The project leverages a wide range of Python libraries to build comprehensive machine learning pipelines:

- *NetworkX* for graph-based data structures and algorithms.
- *PyGraphviz* for graph visualization.
- *Torch* for tensor computations and building neural networks.
- *Scikit-learn* for classical machine learning algorithms.
- *Pandas* for data manipulation and analysis.
- *NumPy* for numerical computations.
- *Matplotlib* for data visualization.
- *Torch-Geometric* for Geometric Deep Learning extensions for PyTorch.

The program encompasses a variety of machine learning models to compare how different models react to the embeddings and representations developed earlier. It includes classical machine learning models from the Scikit-learn library, such as Random Forest, Stochastic Gradient Descent (SGD), and Logistic Regression. These models serve a point of comparison since they don't rely on graph-based data structures and algorithms.

The program also includes Graph Convolutional Network (GCN) models built on top of PyTorch and Torch-Geometric. These models are more complex and powerful, leveraging the graph structure and embeddings to achieve better results. Those models specifically leverage graph-based embeddings and input data, generated using the Node2Vec algorithm to create dense and continuous node features that can be used for subsequent analysis or machine learning tasks.

Main Pipelines: The program is organized around three main pipelines:

1. *GCN Pipeline:* For Graph Convolutional Network models, built using libraries such as Torch and Torch-Geometric.
2. *Classical ML Pipeline:* Leveraging algorithms like Random Forest, SGD, and Logistic Regression from Scikit-learn.
3. *Feature Evaluation Pipeline:* Primarily aimed at evaluating the quality and importance of generated features or graph embeddings.

4.1.3 Other programs

A lot of other programs have been developed for the need of this thesis, but they are not as important as the ones presented above. Those scripts and short program have been developed for several purposes:

- **Data exploration:** Several scripts have been developed to explore the data, and to understand the structure of the heap dump files, the annotations, and the memory graphs.
- **Heap Dump parsing algorithm testers:** Several scripts have been developed to test the heap dump parsing algorithms, and to ensure that the algorithms are working as intended on all possible situations.

- **Graph and embedding generation testers:** Others have been developed to test the graph and embedding generation algorithms, and to ensure that the algorithms are working as intended on all possible situations.

- **Result visualization, analysis and latex generators:** Later in the report, the results will be presented and discussed. Several scripts have been developed to generate the latex tables and plots, and to analyze the results.

All those programs represent a consequent amount of work, and have been invaluable to the success of this thesis. Using CLOC (Count Lines of Code), a command-line utility that can count lines of code in various languages, the following statistics have been obtained:

```
cloc mem2graph research-base predicting_ssh_key_masterarbeit_report
    phdtrack_server_scripts phdtrack_project_3 memory_graph_gcn
    data_processing_masterarbeit --exclude-dir=.venv
```

Listing 4.1: Command used to count the number of lines of code in the *phdtrack* directory, containing the reposiroties of the present thesis.

The following table shows the number of lines of code for each programming language used in the present thesis:

Table 4.1: Code Statistics for Masterarbeit

Language	Files	Blank Lines	Comments	Code Lines
CSV	867	0	0	158305697
Text	25	272	0	50073
Python	132	2326	2682	8050
Rust	50	1343	1149	6823
TeX	28	963	141	6636
Markdown	33	1016	0	2375
JSON	1345	1	0	1677
Jupyter Notebook	2	0	1811	381
Nix	12	50	31	290
TOML	1	2	1	22
Bourne Shell	3	8	8	18
make	1	8	3	18
Dockerfile	1	1	0	2

In the context of this thesis, three programming languages stand out for their specialized roles: Python, Rust, and Nix. Python is predominantly used for the machine learning pipeline, offering ease of use and a rich ecosystem for data science tasks. Rust serves as the backbone for the Mem2Graph program, providing the efficiency required for graph construction and manipulation. Nix is employed for package management and building, ensuring reproducibility across different computing environments. These languages complement each other well, with Python offering high-level abstractions for machine learning, and Rust providing low-level control for performance-critical tasks. Additionally, CSV files are utilized to store model results. TeX is used for generating this report, highlighting the diverse yet complementary set of tools and languages employed in this research.

4.2 Experimental Setup

The experimental setup serves as the backbone of this research, providing a structured framework for conducting large-scale experiments on the server. This section delves into the intricacies of the setup, detailing the steps involved and the tools used. It also includes selected program output logs to offer a granular view of the program parameters, environment, and usage.

The elements discussed in this section are not merely illustrative; they offer invaluable insights into the challenges encountered during the experiments. These details are particularly crucial when discussing the large-scale experiments, as they provide a comprehensive understanding of the various facets involved.

The final experiments were conducted in a systematic manner, following these steps:

1. **Data Cleaning:** The original Zenodo dataset was cleaned to produce a RAW heap dump dataset, serving as the foundational data for the experiments.
2. **Graph and Embedding Generation:** The Mem2Graph Rust program was employed to generate graphs along with their embeddings. A Python launcher script facilitated the generation of multiple memgraph datasets, each with varying combinations of program parameters.
3. **Data Preloading and Validation:** A sanity check Python program was used for data preloading and validation, ensuring the integrity and consistency of the data before proceeding to the experimental phase.
4. **ML and DL GCN Pipelines:** The main Python program was responsible for the seamless launching of data science tasks, machine learning training, and model evaluation. It was designed to cover a predefined range of parameters and model hyperparameters.
5. **Result Collection and Evaluation:** The final step involved the collection, evaluation, and visualization of the results. Error handling mechanisms were in place to make necessary corrections and prepare for the next iteration of experiments. This step also facilitated the confrontation of hypotheses and research questions.

Initial small-scale tests were conducted on a laptop to validate the programs and their results. The final, large-scale experiments were carried out on the Drogon server, equipped with 80 threads and 256 GB of RAM. The computational resources provided by the University of Passau have been invaluable for conducting these experiments, sometimes running for several days straight.

4.2.1 Generation of the memgraph datasets

Using Mem2Graph powerful features, it is possible to generate several memgraph datasets with different parameters. The following command has been used to generate 6 memgraph datasets, each containing 26202 graphs, for a total of 157212 graphs. Those datasets account for different chunk node embeddings and a potential additional filtering feature. The command is run on the Drogon server, with 80 threads.

```
1    [2023-10-24T20:42:44 UTC][INFO mem_to_graph::exe_pipeline::pipeline] OK [t: worker
        -63] [N*202 / 26202 files] [fid: 8683-1650977906]    (Nb samples: 0)
```

```
[2023-10-24T20:42:44 UTC][INFO mem_to_graph::graph_data::heap_dump_data] FILE heap dump raw file path: "/root/phdtrack/phdtrack_data_clean/Performance_Test/V_8_1_P1/32/8794-1650977906-heap.raw"
[2023-10-24T20:42:44 UTC][INFO mem_to_graph::exe_pipeline::pipeline] OK [t: worker-63] [N*203 / 26202 files] [fid: 8794-1650977906]    (Nb samples: 0)
[2023-10-24T20:42:44 UTC][INFO mem_to_graph::exe_pipeline::pipeline] TIME [total pipeline time: 114.84s]
100%|##############################| 6/6 [1:07:15<00:00, 672.62s/it]
Finished! Total time: hours: 1, minutes: 7, seconds: 15 (Drogon Server)
```

Listing 4.2: Sample of final logs of the Mem2Graph program, after generating 6 memgraph datasets from the cleaned heap dump dataset.

It took a little more than an hour to generate 6 memgraph datasets, each containing 26202 graphs. The total number of graphs generated is 157212. The generated memgraph datasets are stored in the data/ directory, as follows:

Figure 4.1: Illustration of the Data Directory Structure

```
/data
├── 0_graph_with_embedding_comments_-v_-a_chunk-header-node_-c_-chunk-semantic-embedding_-e_none_-s_none
├── 1_graph_with_embedding_comments_-v_-a_chunk-header-node_-c_-chunk-statistic-embedding_-e_none_-s_none
├── 2_graph_with_embedding_comments_-v_-a_chunk-header-node_-c_-chunk-start-bytes-embedding_-e_none_-s_none
├── 3_graph_with_embedding_comments_-v_-a_chunk-header-node_-c_-chunk-semantic-embedding_-e_only-max-entropy_-s_activate
├── 4_graph_with_embedding_comments_-v_-a_chunk-header-node_-c_-chunk-statistic-embedding_-e_only-max-entropy_-s_activate
└── 5_graph_with_embedding_comments_-v_-a_chunk-header-node_-c_-chunk-start-bytes-embedding_-e_only-max-entropy_-s_activate
```

As one can see, the dataset directory names are composed with the most important Mem2Graph program parameters, responsible for some feature and embedding generations. The -e flag, short for -entropy-filter is responsible for the filtering using the Shannon entropy, the -s for -chunk-byte-size-filter for chunk byte size filtering. The -c, -graph-comment-embedding-type controls the custom embedding being save alongside each node in the generated .GV memgraph files.

4.2.2 Sanity checking and preloading the generated memgraph datasets

Loading the graph from DOT files into NetworkX graph in Python is a resource intensive operation that consumes all the available computing power on all tested platforms (laptop, servers). It takes several dozens of seconds up to a minute to load a memory graph containing only around 1000 chunk nodes. It has been experimented that saving those loaded graphs using the *pickle* python library allows to perform checks while loading the graph, add more information about each graph. The subsequent retrieval of the graph is much faster, and allows to perform the sanity checks before any further processing. So to verify the memory graph dataset generation as

well as transforming DOT files into pre-saved NetworkX graph, a sanity checking script has been developed.

Below is a sample of the logs generated by the sanity check script:

```
 * Running program...
Passed program params:
param[0]: src/sanity_check_gv_files.py
param[1]: -k
Parsed program params:
keep_old_output: True
skip_dir_starting_with_number: None
dry_run: False
 * Now, performing data loading and sanity checks...
 * Looking for Mem2Graph dataset directories in /root/phdtrack/mem2graph/data...
 * Skipping .gitignore...
 * Found 6 Mem2Graph dataset directories.
[...]
Loading graphs: 100%|##########| 26201/26202 [3:22:17<00:00,  1.17it/s]
Loading graphs: 100%|##########| 26202/26202 [3:22:25<00:00,  3.19s/it]
Loading graphs: 100%|##########| 26202/26202 [3:22:25<00:00,  2.16it/s]
 * Checking embedding length of graphs in /root/phdtrack/mem2graph/data/5_graph_with_
    embedding_comments_-v_-a_chunk-header-node_-c_chunk-start-bytes-embedding_-e_only
    -max-entropy_-s_activate...
-> [x] 26202 graphs in /root/phdtrack/mem2graph/data/5_graph_with_embedding_comments_-
    v_-a_chunk-header-node_-c_chunk-start-bytes-embedding_-e_only-max-entropy_-s_
    activate have been loaded and checked.
-> [_] 0 graphs in /root/phdtrack/mem2graph/data/5_graph_with_embedding_comments_-v_-a
    _chunk-header-node_-c_chunk-start-bytes-embedding_-e_only-max-entropy_-s_activate
    have been skipped (deleted).
[x] 157212 total graphs in the input mem2graph dataset dir paths have been loaded and
    checked.
[_] 0 total graphs in the input mem2graph dataset dir paths have been skipped (deleted
    ).
<END> Program took: 42626.062309 total sec (11h 50m 26s)
```

Listing 4.3: Result logs of the `memory_graph_gcn/src/sanity_check_gv_files.py` program.

The sanity checking file can be considered very long to run, having taken almost 12 hours straight, but it is a necessary step to ensure the quality of the generated memgraph datasets. It is also a good way to check the validity of the generated embeddings, and to ensure that the graphs are re-exported as `NetworkX` graphs that can be loaded much faster than the original DOT files.

4.2.3 Launching the experiments

Two pass of experiments have been conducted, with the exact same parameters and input memgraph dataset. Contrary to expectations, the experiments were much faster on the laptop that on the Drogon server.

- **Time take for the experiments on the laptop:** 12h 31m 53s
- **Time take for the experiments on the Drogon server:** 29h 17m 57s

The experiments have been launched using the following command (here, on Drogon server):

```
nohup python src/main_gcn.py -i /root/phdtrack/phdtrack\_data\_clean/ -p gcn-pipeline
    classic-ml-pipeline feature-evaluation-pipeline -b 6 -a -q -n 16  > output\_ml\_
   2023\_10\_27\_16h\_35.log 2>&1 &
```

Listing 4.4: Command used to launch final experiments, on Drogon server.

Using `nohup` and redirecting the output to a log file allows the experiments to run in the background and enables the retrieval of the output at a later time. The command specifies the input directory containing the annotated DOT (.gv) graph directory with the `-i` flag. All three pipelines are chosen: `gcn-pipeline`, `classic-ml-pipeline`, and `feature-evaluation-pipeline`, as indicated by the `-p` flag. The batch size for parallel processing is set to 6 using the `-b` flag, and the `-a` flag indicates the use of all available Mem2Graph datasets. The `-q` flag enables quiet mode for Node2Vec, and the `-n` flag specifies the use of 16 input graphs.

4.2.4 Dealing with hyperparameter tuning

In the quest to optimize the performance of both ML and DL models, hyperparameter tuning plays a crucial role. This section elaborates on the various strategies and tools employed for hyperparameter tuning in this research.

- **Precise Command Lines for Tuning:** The compute instances and experiment parameters can be finely tuned using precise command-line options. This flexibility allows for a more targeted approach to model optimization, enabling the user to specify various hyperparameters and settings right from the terminal.
- **Python Program for Experiment Launch:** A dedicated Python program has been developed to automate the launching of ML, GCN and FE pipelines. This program takes different hyperparameters as input and initiates the corresponding experiments, thereby streamlining the entire process.
- **Automatic Logging in CSV:** All the results from each experiment, along with the hyperparameters used, are automatically logged into a CSV file. This facilitates easy tracking and comparison of different model configurations and their corresponding performances.
- **Timestamps and Duration Steps:** Each experiment is meticulously logged with precise timestamps and duration steps. This includes the time taken for generating embeddings, as well as the time required for the training and testing phases. Such detailed logging aids in identifying bottlenecks and optimizing the pipeline further.
- **Extensive Experimentation:** Over the course of this research, thousands of experiments have been conducted. These experiments span a wide range of parameters, models, and embeddings, providing a comprehensive understanding of the model behaviors and their sensitivities to different hyperparameters.
- **Automated Result Analysis and Visualization:** To aid in the interpretation of the extensive experimental results, automated scripts have been developed for result analysis and visualization. These scripts generate various plots and metrics that provide insights into the performance and reliability of the different models and configurations.

Of all the parameters, the model types, their respective hyperparameters, the combinations of possible embeddings with their own parameters, and the number of input graphs are the

most important. All those parameters explains the large number of experiments that have been conducted, and the need for a precise and automated way to launch them.

4.2.4.1 Limited number of input graphs due to compute time and memory usage

It also explains why the input number of memgraphs has been limited to 16, as it is already a very large number of experiments to run. Each file taking several dozens of seconds to be transformed into an embedding, this represents around 10 minutes just for the embedding generation phase, for each pipeline. With more than 600 pipelines to run, this already represents dozens of hours of compute time. Depending on the context and model, the training and evaluation phases are also time-consuming, and the more input graphs there are, the longer it takes to train and test the model.

The compute time is not the only issue with dealing with a large number of input graphs. The memory usage is also a problem, as the memory usage increases linearly with the number of input graphs. The parallel processing of only 6 pipelines having 16 memgraphs as inputs generally represents between 16 and 50 GB of RAM usage, depending on the model and the embedding used. Due to this, all tests consisting of trying to run this already limited number of input graphs on the GPU have failed, as the GPU memory is not sufficient to handle the memory usage of the program. The slowness of the CPU alongside memory bandwidth limitations are the main bottlenecks of the program, and the main reasons why the number of input graphs has been limited to 16.

4.2.4.2 Parallel launch of experiments for hyperparameters tuning

To maximize efficiency and expedite the research process, a Python program was developed to launch multiple experiments in parallel. This approach allows for simultaneous testing of various input graphs, machine learning and deep learning models, embedding techniques, and hyperparameters. By leveraging parallel computing, the program significantly reduces the time required for extensive experimentation, thereby accelerating the overall research and development cycle.

Below are the hyperparameters used during the main experiments, as stored in the `hyperparams.json` file:

```
{
   "node2vec_dimensions_range": [128],
   "node2vec_walk_length_range": [16],
   "node2vec_num_walks_range": [50],
   "node2vec_p_range": [0.5, 1.0, 1.5],
   "node2vec_q_range": [0.5, 1.0, 1.5],
   "node2vec_window_range": [10],
   "node2vec_batch_words_range": [8],
   "node2vec_workers_range": [16],
   "randomforest_trees_range": [100, 500, 1000],
   "gcn_training_epochs_range": [20]
}
```

Listing 4.5: JSON hyperparameters used during experiments

Even though the above JSON file shows the hyperparameters selected for the large scale final experiments, pre-experiments have been previously conducted to find some usable values. Due to compute time limitations, the ranges of hyperparameters have been limited to a few values,

and the number of input graphs has been limited to 16. The following JSON file shows the hyperparameters used during the pre-experiments:

```
{
   "node2vec_dimensions_range": [16, 32, 128],
   "node2vec_walk_length_range": [16, 32],
   "node2vec_num_walks_range": [50, 100],
   "node2vec_p_range": [0.5, 1.0, 1.5],
   "node2vec_q_range": [0.5, 1.0, 1.5],
   "node2vec_window_range": [10, 20],
   "node2vec_batch_words_range": [4, 8, 16],
   "node2vec_workers_range": [16, 32],
   "randomforest_trees_range": [100, 500, 1000],
   "gcn_training_epochs_range": [5, 10, 20]
}
```

Listing 4.6: JSON hyperparameters used during experiments

Several observations were made during the early experiments concerning the impact of different hyperparameters on the model's performance, especially concerning the Node2Vec parameters. The following observations were made:

- **Walk Length and Number of Walks:** Increasing the number of walks and the walk length generally improved the model's performance. However, this came at the cost of significantly increased computational time. The benefits plateaued after reaching a certain threshold.
- **Number of Dimensions:** A higher number of dimensions generally led to better results. Lower values were found to be detrimental to the model's performance, indicating the importance of this parameter.
- **Impact of p and q:** The parameters p and q had an unpredictable impact on the model's performance. While some combinations seemed to yield better results, no clear pattern was observed, making these parameters challenging to optimize.
- **Batch Word Range:** The range of batch words had a minimal impact on the model's performance. As a result, an intermediate value was selected for this parameter to balance computational efficiency and performance.

These observations provide valuable insights into the behavior of the models under different hyperparameter settings and serve as a guide for future experiments.

4.2.4.3 Description of the `results.csv` File

The `*results.csv` files are used to store the results of machine learning and deep learning classifiers, including Graph Convolutional Networks (GCNs) and classical classifiers like Random Forest. The files are regular CSVs organized with the following headers:

- **system**: The operating system on which the experiment was run. Here, *Linux*.
- **node_name**: The name of the node in the cluster. Here, either *nixos* (local machine) or *rascoussie* (lab server).
- **release, version**: OS release and version information.

- **machine, processor**: Hardware details.
- **physical_cores, total_cores**: Number of physical and total cores.
- **total_memory, available_memory**: Total and available memory in bytes.
- **start_time**: The start time of the experiment.
- **nb_input_graphs**: Number of input graphs.
- **duration_embedding**: Time taken for embedding.
- **subpipeline_name, index, pipeline_name**: Information about the pipeline model used, with values like *sgd-classifier* or *gcn-with-dropout*.
- **input_mem2graph_dataset_dir_path**: Path to the dataset directory.
- **node_embedding**: Type of node embedding used. Several values are possible, like solo embeddings like *chunk-header-node* or *node2vec*, or combined embeddings like *node2vec-chunk-semantic-embedding*.
- **node2vec_***: Parameters for the Node2Vec algorithm, like the number of walks, the walk length, the window size, the number of dimensions, the number of epochs, and the p and q parameters.
- **random_forest_***: Parameters for the Random Forest algorithm, like the number of estimators (trees) or the number of parallel jobs.
- **imbalance_ratio**: Ratio of the classes in the dataset. For instance, a value of 496.33 means that the dataset contains 496.33 times more negative samples than positive samples. Since no filtering is applied, the ratio is always greater than 1, and generally very high.
- **precision_class_*, recall_class_*, f1_score_class_*, support_class_***: Metrics for each class.
- **true_positives, true_negatives, false_positives, false_negatives**: Confusion matrix elements.
- **AUC**: Area under the ROC curve.
- **duration_train_test**: Time taken for training and testing.
- **nb_node_features**: Number of node features. This value directly depends on the node embedding used, and the input memory graph dataset parameters used during generation.
- **first_gcn_training_epochs**: Number of epochs in GCN training phase.

Each row in the `*results.csv` files represents a single experiment run with a specific configuration and its corresponding results. Keeping a precise track of the parameters used for each experiment is crucial for reproducibility and traceability. They also form the basis for the analysis and visualization of the results.

4.3 Obtained Results

This section presents a comprehensive overview of the results obtained with the final experimentation. The outcomes are detailed in multiple formats, including correlation matrices, performance metrics tables, and visualizations, to provide a precise understanding of the model performances. The aim is to elucidate the effectiveness of different features and embeddings in the context of our machine learning pipelines. In-depth discussions on these results will be reserved for the following "Discussions" part.

4.3.1 Feature Engineering results

This section delves into the results obtained from the feature engineering efforts carried out during the experiments. The results are presented in multiple correlation matrices.

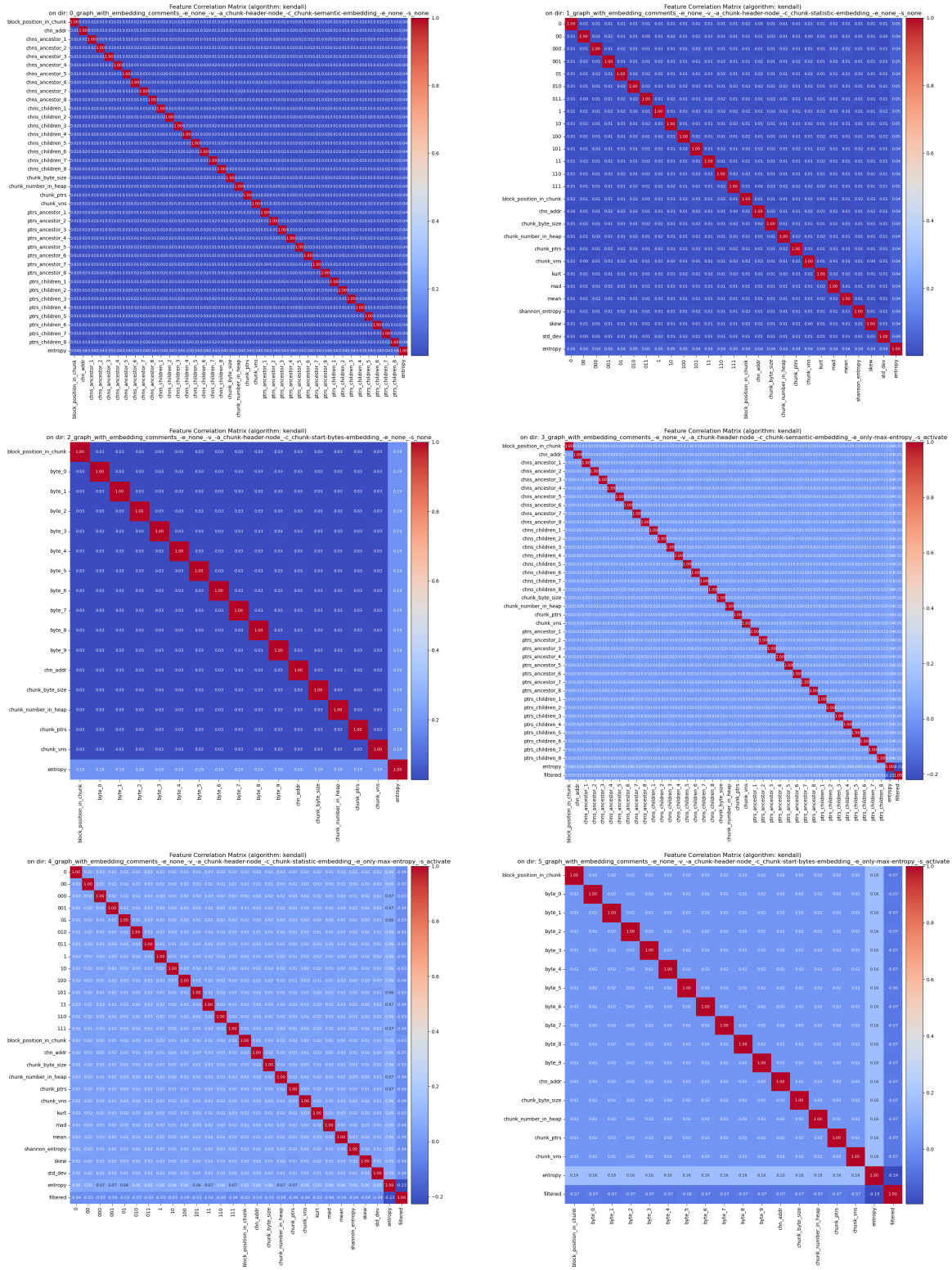


Figure 4.2: Feature correlation matrices on the different Mem2Graph-generated datasets. Used algorithm: Kendall.

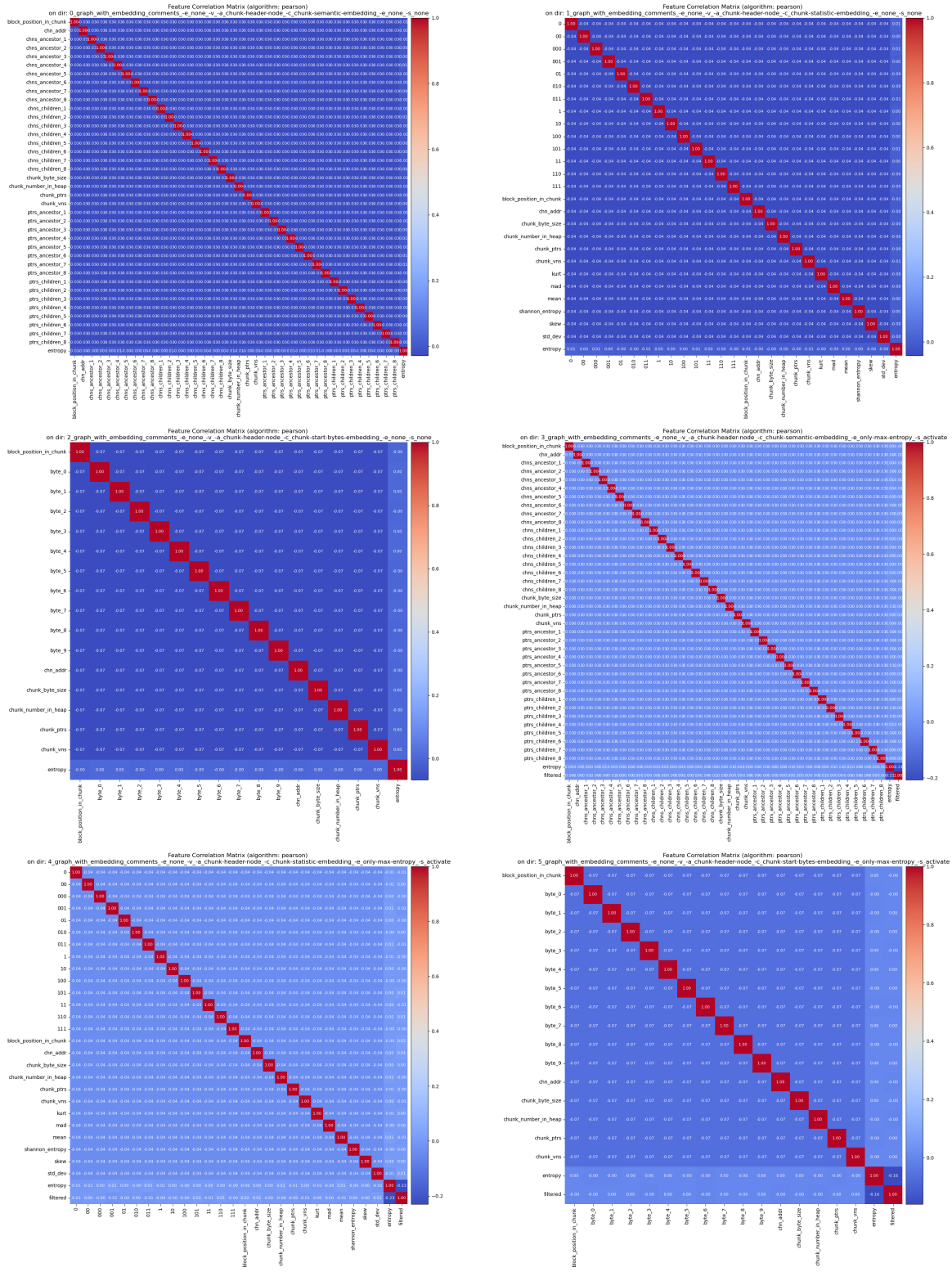


Figure 4.3: Feature correlation matrices on the different Mem2Graph-generated datasets. Used algorithm: Pearson.

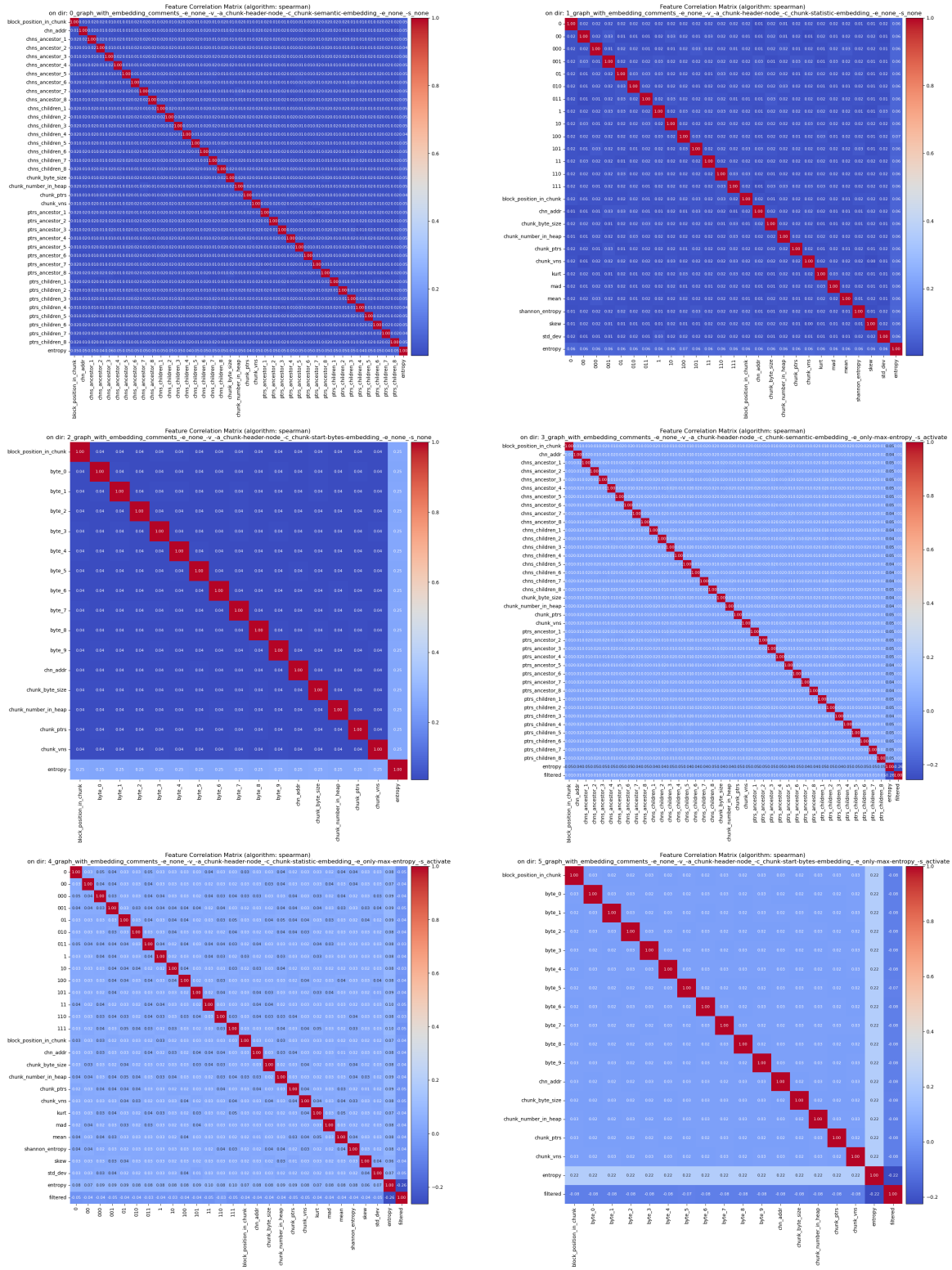


Figure 4.4: Feature correlation matrices on the different Mem2Graph-generated datasets. Used algorithm: Spearman.

4.3.2 Classic Model results

Tables are provided to summarize the performance of different pipelines and models. These tables include four classical machine learning metrics: precision, recall, F1 score, and the Area Under the Curve (AUC). Each table offers a snapshot of how well each model performs on the key chunk

prediction task.

Table 4.2: Best instances of model: logistic-regression.

Best at	Precision	Recall	F1 Score	AUC
precision	1.0000	0.0417	0.0800	0.5208
recall	0.3333	0.5000	0.4000	0.7471
f1 score	0.3333	0.5000	0.4000	0.7471
AUC	0.2449	0.5000	0.3288	0.7486

Table 4.3: Best instances of model: random-forest.

Best at	Precision	Recall	F1 Score	AUC
precision	1.0000	0.0417	0.0800	0.5208
recall	1.0000	0.0833	0.1538	0.5417
f1 score	1.0000	0.0833	0.1538	0.5417
AUC	1.0000	0.0833	0.1538	0.5417

Table 4.4: Best instances of model: sgd-classifier.

Best at	Precision	Recall	F1 Score	AUC
precision	1.0000	0.0417	0.0800	0.5208
recall	0.4615	1.0000	0.6316	0.9962
f1 score	0.4615	1.0000	0.6316	0.9962
AUC	0.4615	1.0000	0.6316	0.9962

4.3.3 Deep Learning GCN Model results

Best models obtained after the hyperparameter search, on a range of embeddings and models, this time focusing on the GCN models.

Table 4.5: Best instances of model: very-simple-gcn.

Best at	Precision	Recall	F1 Score	AUC
precision	0.6000	0.5000	0.5455	0.7489
recall	0.2609	1.0000	0.4138	0.9907
f1 score	0.6000	0.5000	0.5455	0.7489
AUC	0.2609	1.0000	0.4138	0.9907

Table 4.6: Best instances of model: simple-gcn.

Best at	Precision	Recall	F1 Score	AUC
precision	0.5000	0.5000	0.5000	0.7484
recall	0.2308	1.0000	0.3750	0.9891
f1 score	0.5000	0.5000	0.5000	0.7484
AUC	0.2609	1.0000	0.4138	0.9907

Table 4.7: Best instances of model: first-gcn.

Best at	Precision	Recall	F1 Score	AUC
precision	0.5000	0.5000	0.5000	0.7484
recall	0.2727	1.0000	0.4286	0.9913
f1 score	0.5000	0.5000	0.5000	0.7484
AUC	0.2727	1.0000	0.4286	0.9913

Table 4.8: Best instances of model: gcn-with-dropout.

Best at	Precision	Recall	F1 Score	AUC
precision	0.3500	0.2333	0.2800	0.6152
recall	0.0863	0.8000	0.1558	0.8810
f1 score	0.2110	0.7667	0.3309	0.8767
AUC	0.0863	0.8000	0.1558	0.8810

Table 4.9: Best instances of model: advanced-gcn.

Best at	Precision	Recall	F1 Score	AUC
precision	0.2097	0.4333	0.2826	0.7129
recall	0.0533	0.9000	0.1006	0.8943
f1 score	0.2097	0.4333	0.2826	0.7129
AUC	0.1552	0.9000	0.2647	0.9390

4.4 Compared Performances of models and embeddings

In our experiments, we limited the analysis to a maximum of 16 input graphs, which may appear to be a relatively low number. However, this limitation was necessary due to the extensive range of hyperparameters, embeddings, and models that we aimed to evaluate. Despite this constraint, we were able to run a total of 7976 machine learning pipelines, accumulating over 100 hours of compute time. This extensive computational effort underscores the complexity and depth of the feature engineering and model evaluation processes undertaken in this study. The following tables and visualizations provide a comprehensive overview of the results obtained from these experiments.

Table 4.10: Results for the model very-simple-gcn

Model	Best Precision	Best Recall	Best F1 Score	Best AUC
advanced-gcn	0.2097	0.9000	0.2826	0.9390
first-gcn	0.5000	1.0000	0.5000	0.9913
gcn-with-dropout	0.3500	0.8000	0.3309	0.8810
logistic-regression	1.0000	0.5000	0.4000	0.7486
random-forest	1.0000	0.0833	0.1538	0.5417
sgd-classifier	1.0000	1.0000	0.6316	0.9962
simple-gcn	0.5000	1.0000	0.5000	0.9907
very-simple-gcn	0.6000	1.0000	0.5455	0.9907

In addition to the tabular data, we also offer visualizations to facilitate a more intuitive comparison between models and embeddings. These graphical representations aim to make the complex

data more digestible and provide insights that may not be immediately obvious from the tables alone.

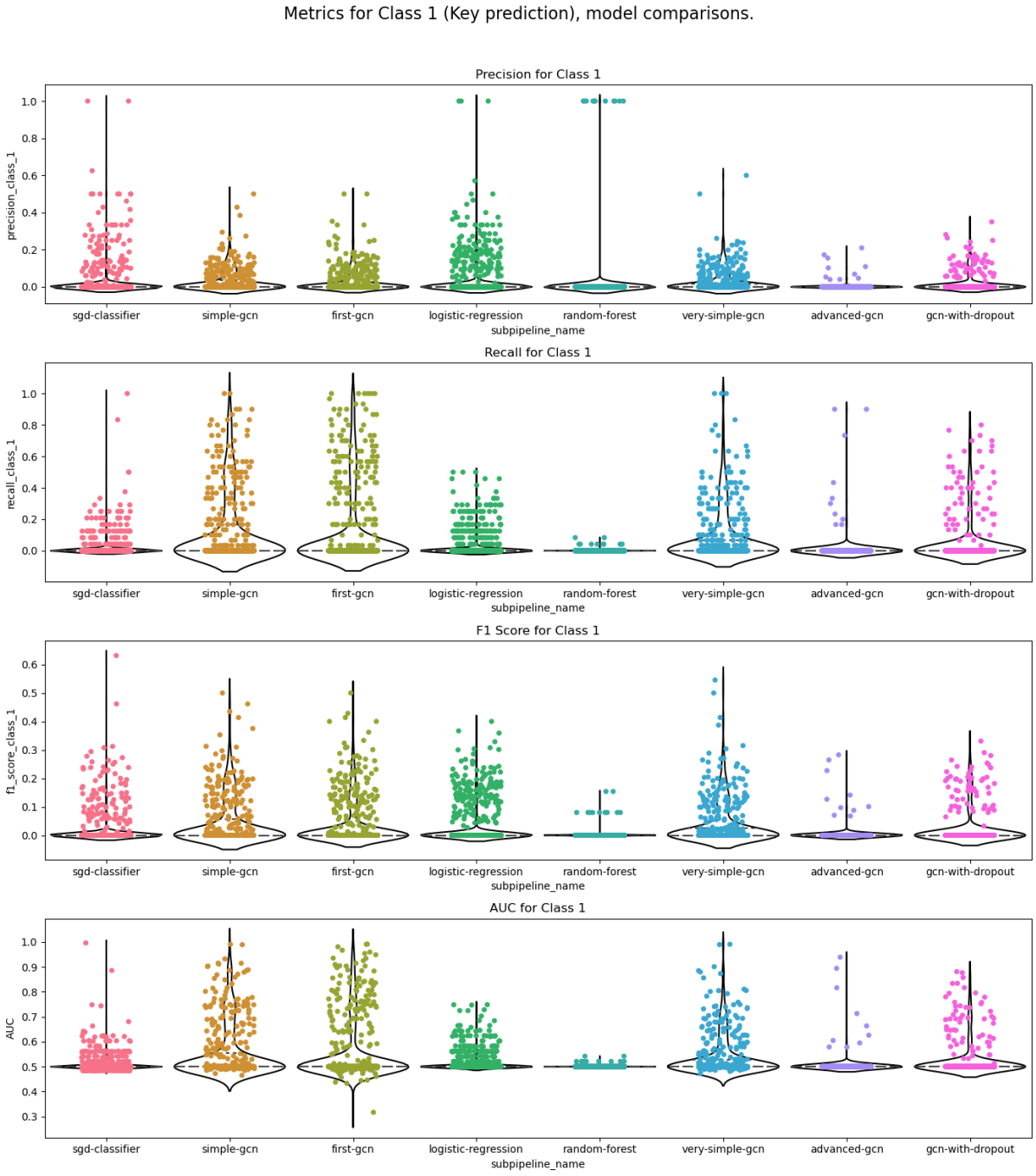


Figure 4.5: Visualization of the result metrics use to compare model performance on memory graphs, for different embeddings and hyperparameters.

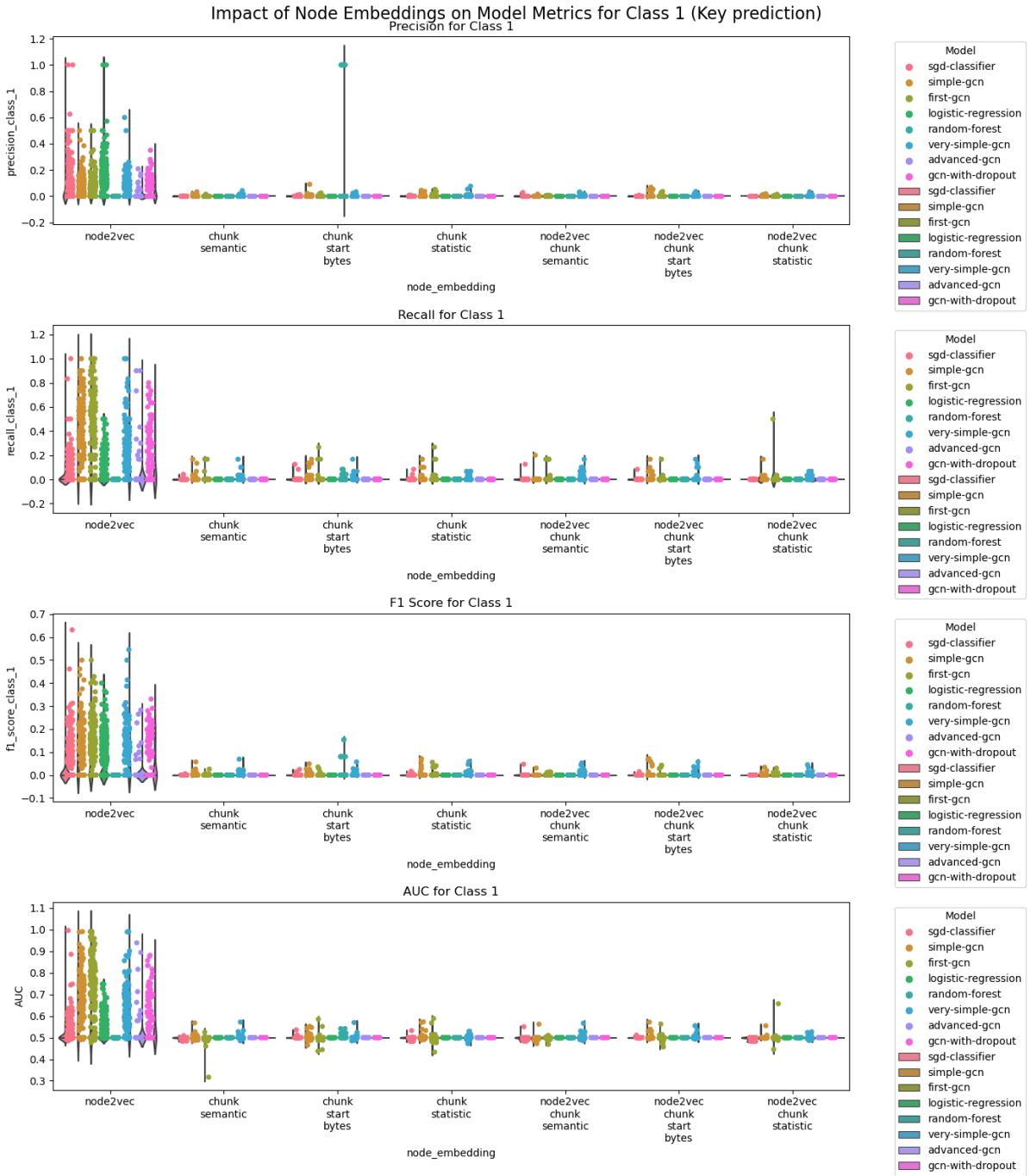


Figure 4.6: Visualization of the result metrics use to compare model performance per memory graph node embedding strategies.

It’s worth noting that while this section provides a detailed account of the results, an in-depth discussion about these findings, their implications, and potential future work will be covered in the following "Discussions" section.

5 Discussion

In the previous chapter, the results of the experiments were presented. This chapter aims to provide an in-depth discussion of those results, as well as to identify the limitations of the experiments and to propose avenues for future research.

5.1 Discussion of the results

The following subsections will discuss the results obtained in the experiments, and will provide insights into the performance of the different models, as well as the impact of the different features and embeddings on the performance of the models.

5.1.1 Objectives of the experiments

The primary objectives of the experiments conducted in this study are multi-faceted. First and foremost, we aim to demonstrate the feasibility of utilizing machine learning and deep learning models to predict chunks with keys in the OpenSSH program based on a graph-like representation of the heap dump files provided in the original dataset. To achieve this, we employ a range of algorithms to extract features from memory graphs, or 'memgraphs'. These algorithms include not only custom solutions tailored to our specific needs but also well-established, powerful algorithms like Node2Vec. Furthermore, we seek to evaluate the impact of these diverse features on the performance metrics of the predictive models. Lastly, we compare the performances of various models to identify the most effective approaches for our specific use-case.

5.1.2 Discussing features and embeddings

In this section, we delve into the intricacies of the features and embeddings used in our experiments, focusing particularly on their interrelationships as revealed by correlation matrices. Correlation matrices provide a quantitative measure of how different features of custom embeddings relate to each other. Each cell in the matrix represents the correlation coefficient between two features, which ranges from -1 to 1. A high positive value indicates a strong positive correlation, meaning that as one feature increases, the other tends to also increase. A negative value would indicate the opposite.

It's worth noting that performing this analysis on Node2Vec embeddings is generally considered irrelevant. Node2Vec embeddings are designed to capture the neighborhood structure of nodes in a way that is optimized for machine learning tasks, and their dimensions do not have an easily interpretable meaning. Therefore, analyzing the correlation between different dimensions of a Node2Vec embedding is unlikely to provide insights that are useful for feature engineering.

To interpret the correlation matrices, we use a color-coded system where red signifies high correlation and blue signifies low or no correlation. In the context of machine learning, understanding feature correlation is crucial for several reasons:

- **Feature Selection:** Highly correlated features carry redundant information, which may not only be unnecessary but can also lead to overfitting and poor generalization.

- **Interpretability:** Understanding how features are correlated can provide insights into the underlying structure of the data and the problem being solved.
- **Computational Efficiency:** Eliminating correlated features can reduce the dimensionality of the problem, making the model simpler and faster to train.

Therefore, the correlation matrices serve as a valuable tool for both feature selection and model interpretation. In that context, and looking at the correlation matrices provided for the different algorithms like Pearson, Spearman, and Kendall correlation, we can see that the correlation between the different features is generally very low, meaning that the features are not correlated to each other. This is a good thing, as it means that the features are not redundant, and that they are all bringing new information to the model. No matter the correlation algorithm used, the matrices look very similar, and the correlation between the features is very low. The only features that stands a bit from the rest are the features corresponding to the filtering and entropy. This is actually just a sign that the entropy was indeed used in the filtering algorithm, since key chunks are generally more entropic than non-key chunks. In practice, that's also why the experiments have been run with and without this filtering feature.

5.1.3 Classic ML models performances

In this subsection, we discuss the performance of three tested classical binary classification machine learning models, namely Logistic Regression, Random Forest, and SGD Classifier, in the context of key chunk prediction. The models were evaluated based on four key metrics: Precision, Recall, F1 Score, and AUC (Area Under the Curve).

5.1.3.1 Logistic Regression

The Logistic Regression model excels in precision with a perfect score of 1.0000 but falls short in recall, F1 score, and AUC. The model is highly precise but fails to capture the majority of the positive instances, as indicated by the low recall of 0.0417. This suggests that while the model makes very few false-positive errors, it misses a large number of true positives.

5.1.3.2 Random Forest

Random Forest shows excellent precision at the expense of recall. It has a high precision of 1.0000 but a very low recall of 0.0833, indicating that it is precise but not sensitive. The AUC of 0.5417 suggests that the model's ability to distinguish between the classes is slightly better than random guessing.

5.1.3.3 SGD Classifier

The SGD Classifier stands out in terms of recall and AUC, both scoring close to a perfect score This indicates that the model is excellent at identifying all the positive instances and distinguishing between the two classes. However, its precision isn't always perfect, suggesting a higher rate of false positives.

5.1.3.4 Comparison

Upon comparing the three models, it's evident that each has its own strengths and weaknesses. Logistic Regression has a much better overall recall than Random Forest which is very precise but fails to find a lot of cases. However, SGD Classifier is clearly the best overall model here, since it is highly sensitive and excellent in class separation but lacks in precision.

In summary, the SGD Classifier would be more appropriate in the binary classification task of predicting is a chunk is a key chunk or not. Random Forest offers a balanced but mediocre performance and could serve as a baseline model. Those models merely serve as a comparison point for the deep learning models.

5.1.4 GCN models performances

In this subsection, we delve into the performance metrics of five different Graph Convolutional Networks (GCN) models: Very Simple GCN, Simple GCN, First GCN, GCN with Dropout, and Advanced GCN. These models were evaluated on the same four key metrics as the classical models: Precision, Recall, F1 Score, and AUC.

5.1.4.1 Very Simple GCN

The Very Simple GCN model shows generally balanced performances. It's worth noting that the best instance of the model reached a perfect recall of 1.0000, indicating excellent sensitivity. However, the precision of the model is at best only of 0.6000, suggesting a higher rate of false positives. Its best instance having an AUC score of 0.9907 suggests excellent class separation capabilities.

5.1.4.2 Simple GCN

This model has similar or slightly lower performance metrics to the Very Simple GCN, with a precision and recall of 0.5000. The AUC score is slightly lower at 0.9891 but still indicates excellent class separation.

5.1.4.3 First GCN

The First GCN model also looks very similar to the two previous models. Its best AUC instance has a high AUC indicating excellent sensitivity and class separation.

5.1.4.4 GCN with Dropout

This model has the lowest precision among the GCN models at 0.3500 but shows a decent AUC score of 0.8810. The model overall seems to be more sensitive, but less precise compare to the simpler models introduced before. This was indeed expected due to the use of dropout, which is known to increase sensitivity at the expense of precision.

5.1.4.5 Advanced GCN

The Advanced GCN model, despite its complexity, does not outperform the simpler models in this limited dataset. It displays the worst performances in precision, f1 score and AUC, but has the best recall of all the GCN models, at 0.9000. This suggests that the model is very sensitive but not very precise, which is not surprising considering the complexity of the model, and the limited number of input memgraphs. This trend is likely to change if tested with a larger number of training graphs.

5.1.4.6 Layer Complexity

It's worth noting that the simpler models (Very Simple and Simple GCN) with only 2 layers tend to perform better in this limited training dataset. This could be due to the low number of example graphs, which might not be sufficient to train the more complex, 4-layer Advanced GCN model effectively. Yet, the Advanced GCN model has the best recall of all the GCN models, suggesting that it is more sensitive than the simpler models.

In summary, each GCN model has its own set of strengths and weaknesses. While some excel in precision, others are more sensitive or better at class separation. The choice of model would depend on the specific requirements of the task at hand.

5.1.5 Comparing the embeddings impact on the models

Based on the results of the experiments, it's evident that the choice of embedding has a significant impact on the performance of the models. As this can be seen in 4.4, and surprisingly, it seems that the custom embeddings developed specifically for this research are actually not the best performing embeddings. As such, adding the features from those custom embeddings to the Node2Vec embeddings does not improve the performances of the models, but actually degrade them. This is a very surprising result. The Node2Vec embedding is actually the best performing embedding, no matter the model used, which is even more surprising, since the GCN models were actively using the graph structure, which is not the case for the classic ML models. This means that the Node2Vec embedding, which is purely based on the graph structure, is actually the best performing embedding, and that the custom embeddings are not bringing any additional information to the models. Thus, this demonstrates that the memgraph structure is in and on itself sufficiently meaningful to perform the classification task.

It's worth mentioning the master thesis of Clément Lahoche that actually dives deeper in the question of embedding quality and impacts. His work displays significantly different results than the ones obtained in this research about embeddings. It shows that the question of embeddings is indeed a complex topic, and that the results can vary a lot depending on the dataset, the model, and the approach used.

The big difference between this present work and his thesis is that he used a much larger number of input memgraphs, but also that he actually perform a real rebalancing on the dataset before training. This means that whereas the experiments perform in this work have been dealing with a very high imbalance rate, his classifiers have been trained on a much better balanced dataset. This is a huge difference, and it's not surprising that the results are so different. The reason why this thesis did not use active rebalancing is that we wanted to explore the impact of learning on a full memgraph dataset, without alteration of those memgraphs. This is due to the

fact that this work has been focused around those graphs and their equivalent in the classification phase with GCNs that rely on the graph structure. The goal was to see if the graph structure was enough to perform the classification, and the results are very promising. Results show that the imbalance ratio is not a problem for the models, as they are able to perform well despite the very high imbalance ratio.

5.1.6 Comparing GCN and classical ML models

In this subsection, we aim to compare the performance of classical machine learning models with Graph Convolutional Networks (GCNs) for the task of key chunk prediction. The comparison is based the results of the experiments, the best instances of each model on each metric, but also the distribution of the metrics as illustrated in 4.4.

5.1.6.1 Overall Performance

For overall balanced performance, the SGD Classifier from the classical models and the Very Simple GCN from the GCN models stand out. The SGD Classifier excels in recall and AUC, making it highly sensitive and excellent in class separation, although it lacks in precision. On the other hand, the Very Simple GCN shows balanced metrics across the board. It has a decent recall and AUC, but its precision is not perfect. However, it is the best performing GCN model in terms of precision.

5.1.6.2 Precision

If precision is the primary concern, then Logistic Regression and Random Forest from the classical models and the Very Simple GCN from the GCN models are the best choices. Logistic Regression and Random Forest both have a perfect precision score of 1.0000, while the Very Simple GCN has a precision of 0.6000, which is the highest among the GCN models.

5.1.6.3 Recall

For applications where high recall is crucial, the SGD Classifier from the classical models and the Advanced GCN from the GCN models are the most suitable. The SGD Classifier has a near-perfect recall, while the Advanced GCN has the highest recall among the GCN models at 0.9000.

5.1.6.4 Class Separation (AUC)

If the ability to distinguish between classes is of utmost importance, then the SGD Classifier from the classical models and the Very Simple GCN from the GCN models are the best options. Both models have AUC scores close to 1, indicating excellent class separation capabilities, between key and non-key chunks.

5.1.6.5 Considerations for Small Datasets

It's worth noting that the simpler GCN models (Very Simple and Simple GCN) tend to perform better on the limited dataset used for training. This suggests that for small datasets, simpler models may be more effective. The Advanced GCN model, despite its complexity, does not perform as well but shows promise in terms of high recall, which could potentially improve with more training data.

5.1.6.6 Summary

In summary, the choice of model would depend on the specific metric that is most important for the task. For balanced performance, the SGD Classifier and Very Simple GCN are recommended. For high precision, Logistic Regression or Very Simple GCN should be considered. For high recall, the SGD Classifier or Advanced GCN would be the most appropriate. Finally, for excellent class separation, the SGD Classifier and Very Simple GCN are the best choices according to the experiments.

5.2 Limitations of the Experiments

While the experiments conducted offer valuable insights into the performance and capabilities of the machine learning and deep learning classifiers, it is crucial to acknowledge the limitations that were inherent in the experimental setup. These limitations range from computational resources to the scale and duration of the experiments. Understanding these constraints is essential for interpreting the results accurately and for identifying avenues for future research. This section aims to discuss the following limitations in detail:

- **Number of Compute Instances:** The experiments were constrained by the available number of compute instances, affecting the scale at which they could be conducted.
- **Number of Input Graphs:** The quantity of input graphs used in the experiments was rather limited, which could impact the generalizability of the results.
- **Duration of the Experiments:** The time allocated for each experiment was finite, potentially affecting the depth of the analysis.
- **CPU-only Environment:** The experiments were conducted in a CPU-only setting, without the acceleration benefits that a GPU could offer, due to problems of memory consumption being too high for the GPU. Additional work on this aspect could significantly improve the performance of the experiments, especially the Node2Vec embedding generation.
- **High Memory Bandwidth Usage:** The experiments were characterized by high memory bandwidth usage, which could have implications for performance.

The subsequent subsections will delve into each of these limitations, providing a comprehensive understanding of their impact on the experiments.

5.2.1 Limited number of input graphs

The number of input graphs is a parameter that can be easily changed, and the experiments can be run again with a higher number of input graphs, but it would take a very long time to run, and the results would be similar although improved to the ones obtained with 16 input graphs. Improving the performances could be done essentially by recoding the program Node2Vec embedding part and adding those results directly inside the *Mem2Graph* program. Leveraging the Rust zero-abstraction costs philosophy, it would be possible to improve the performances by a probable factor of 100 to 1000 times, and to run the experiments with a higher number of input graphs. For ease of development and handling of the results, I would still recommend to perform the machine learning related experiments using the powerful Python dedicated libraries.

That being said, it's remarkable that the models can perform so well considering the very small number of training memgraph and the very high imbalance ratio of the dataset. The imbalance ratio is the ratio of the number of negative samples over the number of positive samples. In the case of the dataset used in this research, the imbalance ratio is very high, ranging generally at several hundred times more negative samples than positive samples. No rebalancing has been performed on the dataset since we wanted to explore the impact of learning on a full memgraph dataset, without alteration of those memgraphs.

The results obtained in this research are already very promising, and the imbalance ratio is not a problem for the models, as they are able to perform very well despite the very high imbalance ratio.

5.2.2 Duration of the experiments

Although a lot of efforts have been put to deal both with dataset reduction, for instance transforming the initial block address prediction into a chunk address prediction problem, then optimizing a lot of computing through the use of a dedicated Rust parallel and optimized program, then using techniques like file preloading, the sheer number of hyperparameters and the number of experiments to run, as well as the compute time for the Node2Vec embedding generation, make the experiments very long to run.

Below are the duration times for the different steps of the experiments.

Table 5.1: Duration times for duration of embedding generation in ML/DL/FE pipeline (in seconds).

Model	**Min duration**	**Max duration**	**Min duration**
advanced-gcn	506.5721079074733	1548.909129	0.129933
first-gcn	503.49931116140345	1548.909129	0.129933
gcn-with-dropout	506.5721079074733	1548.909129	0.129933
logistic-regression	505.3690870955711	1565.660571	0.06828
random-forest	505.3690870955711	1565.660571	0.06828
sgd-classifier	505.3690870955711	1565.660571	0.06828
simple-gcn	506.5721079074733	1548.909129	0.129933
very-simple-gcn	506.5721079074733	1548.909129	0.129933

Considering the tested models are not especially complex, and since the number of input memgraph stays limited, the duration of the training and testing steps is actually quite small:

Table 5.2: Duration times for duration of training and testing in ML/DL/FE pipeline (in seconds).

Model	**Min duration**	**Max duration**	**Min duration**
advanced-gcn	10.843279690391459	496.664757	2.247229
first-gcn	5.5063934491228075	279.007307	0.583914
gcn-with-dropout	8.272014035587189	418.445809	0.905496
logistic-regression	1.3495811305361307	4.165722	0.362695
random-forest	11.72722453030303	48.723031	0.315739
sgd-classifier	0.6751382750582751	6.405859	0.020952
simple-gcn	4.337255024911032	163.587265	0.509536
very-simple-gcn	8.188304871886121	58.307836	0.242535

All those values have been produced only by the python pipeline program. The embedding time is actually mostly accounting for the Node2Vec generation, since the other embeddings are already loaded in memory as they are produced and included in the outputs of Mem2Graph. The Node2Vec generation is the most time-consuming part of the pipeline, and it is the reason why the experiments take so long to run. Transferring this algorithm into Mem2Graph would be a huge improvement, and would allow to run the experiments with a much higher number of input memgraphs.

Throughout this report, we have introduced a lot of concepts, diverse algorithms, and different models. The experiments conducted in this research were limited in scope due to the focus around a large set of models, embeddings and hyperparameters which already represented a consequent amount of work and computational resources. However, there are many more avenues for future research, which will be introduced with the conclusion of this report.

6 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 Knowledge 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.

6.1 Summary of Results

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

6.1.1 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 around 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 consists 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.

However, its worth noting that instead of doing active node filtering and data rebalancing, we have run the experiments and model training and evaluation on full memory graphs with high imbalance ratio. This is because we wanted to test GCN that are able to handle imbalanced data with graphs of varying size, and because we wanted to test the ability of the GCN to learn from the imbalanced data.

6.1.2 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, or memgraph. This graph is a data 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. Those kinds of graphs are only composed of Chunk Header Nodes whose address is considered the address of the related chunk. This allows to reduce the size of the graph by a factor of 10 to 100.

6.1.3 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 doesn't have the same results on the ML and GCN models, depending on the strengths and weaknesses of the different models.

However, it's worth noting that the best results are always obtained when using the Node2Vec embedding, no matter the type of model used. These observations are likely to be due to the fact that Node2Vec is able to capture the structure of the graph, and that the structure of the graph is the most important feature for the models, given a relatively small number of input memgraph and considering that those memgraphs are highly imbalanced.

6.1.4 Conclusion on Model Performances

In this study, we compared the efficiency of classical machine learning models and Graph Convolutional Networks (GCN) in the task of key chunk prediction. Our findings indicate that the choice of model largely depends on the specific metric of interest. For a balanced performance encompassing recall, precision, and AUC, the SGD Classifier from the classical models and the Very Simple GCN from the GCN models are the most promising.

If precision is the primary metric of concern, Logistic Regression and Random Forest from the classical models excel with perfect scores, while the Very Simple GCN leads among the GCNs. For scenarios where high recall is crucial, the SGD Classifier and the Advanced GCN model stand out. Both models also perform exceptionally well in class separation, as indicated by their high AUC scores.

It's also worth noting that simpler GCN models like the Very Simple and Simple GCN tend to perform better on limited datasets, suggesting their suitability for tasks with constrained data

availability. In contrast, more complex models like the Advanced GCN show promise in terms of high recall but require more extensive training data for optimal performance.

In summary, the optimal model selection is contingent upon the specific requirements of the task, whether it be balanced performance, high precision, high recall, or excellent class separation. The choice of model also depends on the availability of training data. For instance, if the dataset is limited, a simpler GCN model like the Very Simple GCN is preferable. However, if the dataset is extensive, a more complex model like the Advanced GCN is more suitable.

6.2 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.

Appendix

.1 Code and files

```
strict digraph "17016-1643962152" {
   "CHN(0x558343d21d40)" [label="CHN(1)" color="cyan" style=filled shape=square];
   "CHN(0x558343d1a448)" [label="CHN(2)" color="cyan" style=filled shape=square];
   "VN(0x558343d1a450)" [label="VN" color="grey" style=filled];
   "VN(0x558343d1a458)" [label="VN" color="grey" style=filled];
   "PN(0x558343d24ae8)" [label="PN" color="orange" style=filled shape=hexagon];
   "KN_KEY_A(0x558343d29460)" [label="KN(A)" color="green" style=filled];
   "KN_KEY_B(0x558343d2b960)" [label="KN(B)" color="green" style=filled];
   "CHN(0x558343d21d40)" -> "KN_KEY_A(0x558343d29460)" [label="dts" weight=1]
   "PN(0x558343d204e8)" -> "KN_KEY_A(0x558343d29460)" [label="ptr" weight=1]
   "CHN(0x558343d21d40)" -> "KN_KEY_B(0x558343d2b960)" [label="dts" weight=1]
   "PN(0x558343d2deb8)" -> "KN_KEY_B(0x558343d2b960)" [label="ptr" weight=1]
   "CHN(0x558343d21d40)" -> "KN_KEY_C(0x558343d29080)" [label="dts" weight=1]
   "PN(0x558343d204e0)" -> "KN_KEY_C(0x558343d29080)" [label="ptr" weight=1]
   "PN(0x558343d24ae8)" -> "VN(0x558343d1a010)" [label="ptr" weight=1]
   "PN(0x558343d1a240)" -> "VN(0x558343d20680)" [label="ptr" weight=1]
}
```

Listing 1: The DOT file of uncompressed block memgraph, here *Training/basic/V_7_1_P1/24/17016-1643962152-heap.raw*, with real addresses. Output is cropped.

.2 Memory Graphs

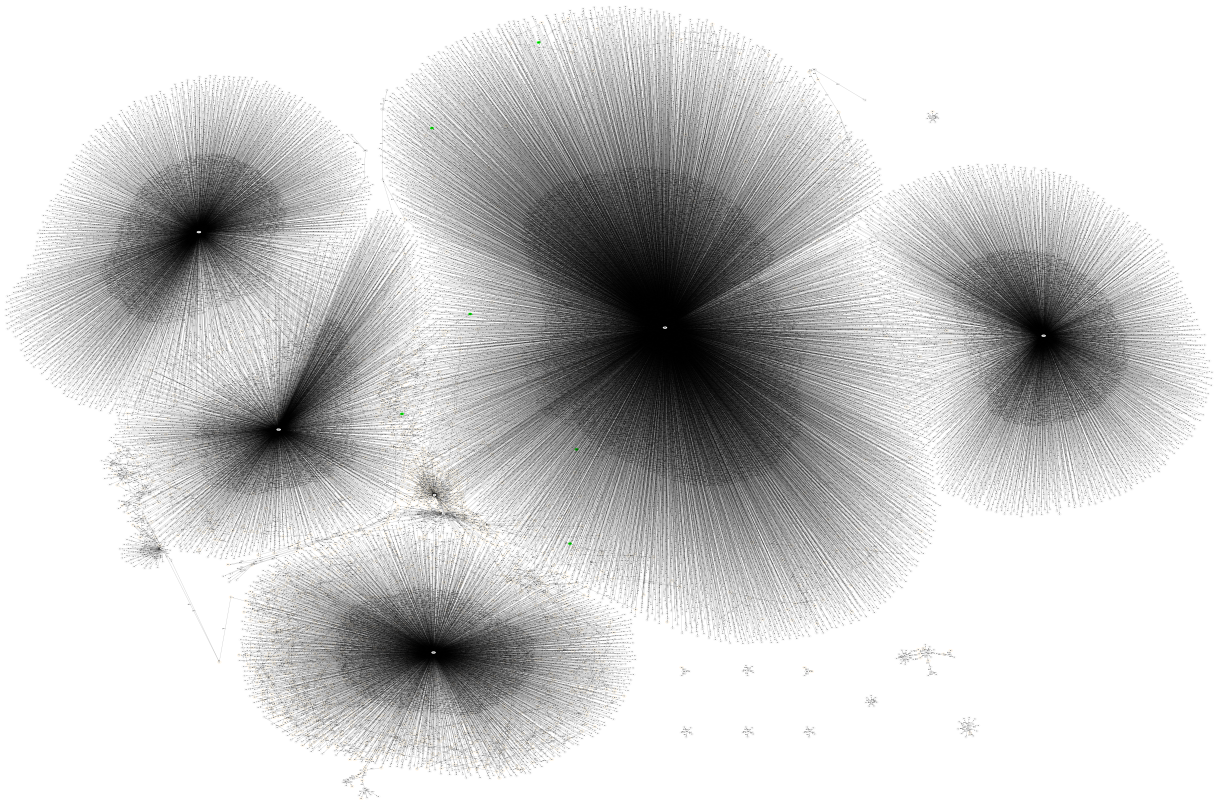


Figure 1: Visualization of the full memory graph generated from *Training/scp/V_7_8_P1/16/302-1644391327-heap.raw*.

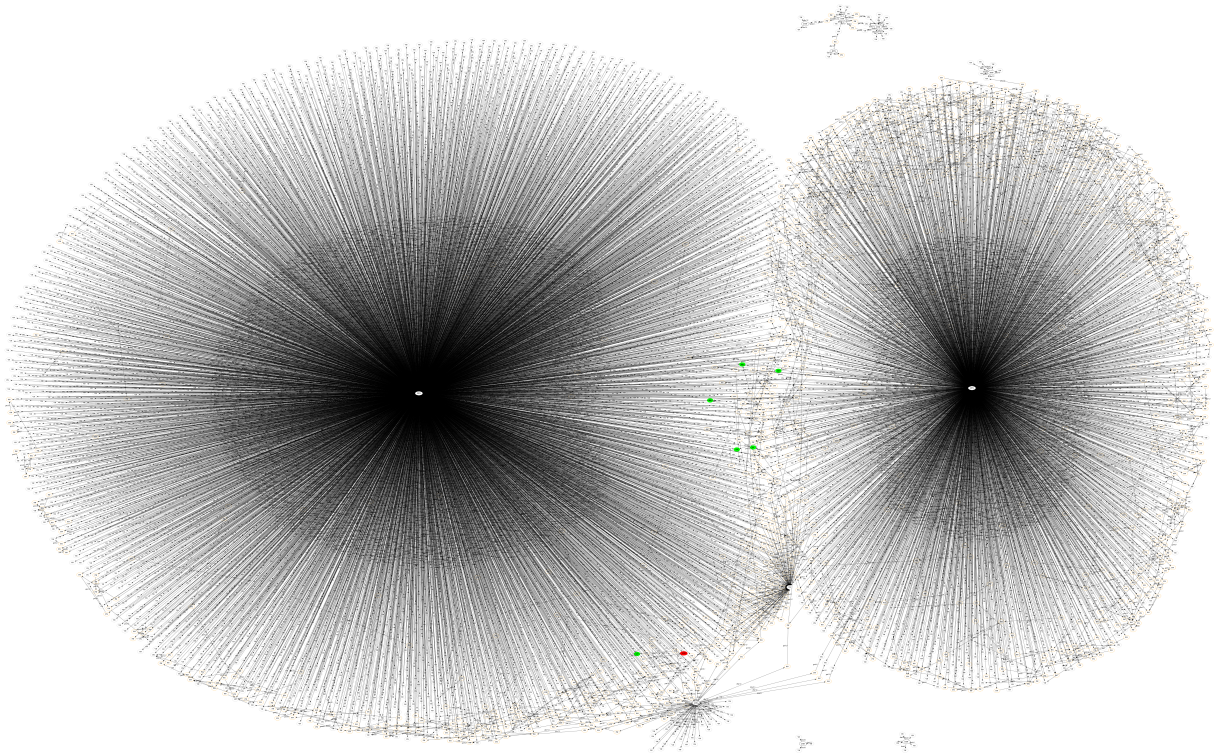


Figure 2: Visualization of the full memory graph generated from *Training/basic/V_7_1_-P1/24/17016-1643962152-heap.raw*.

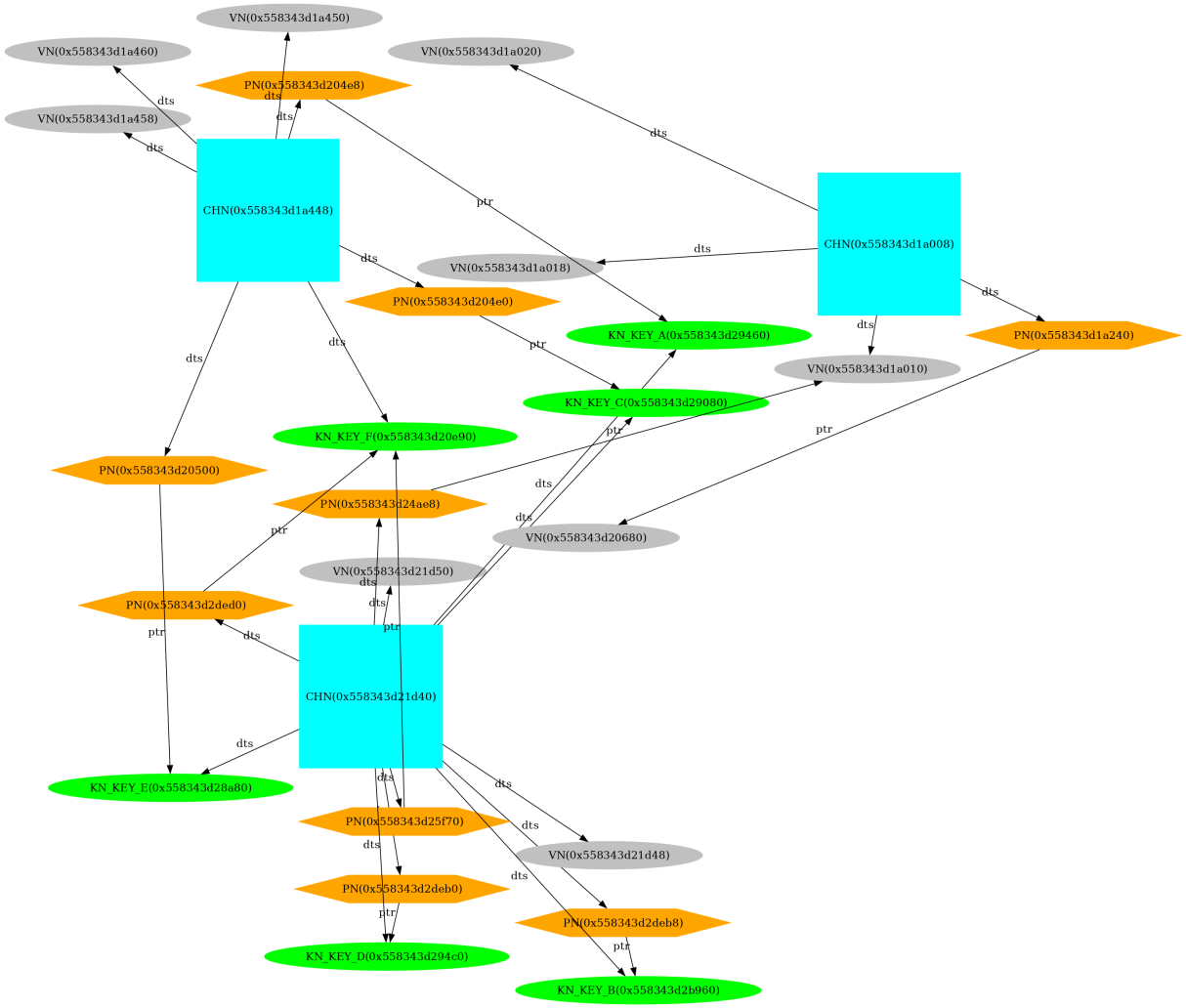


Figure 3: Visualization of a truncated memory graph generated from *Training/basic/V_7_1_-P1/24/17016-1643962152-heap.raw*. Here with real addresses.

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

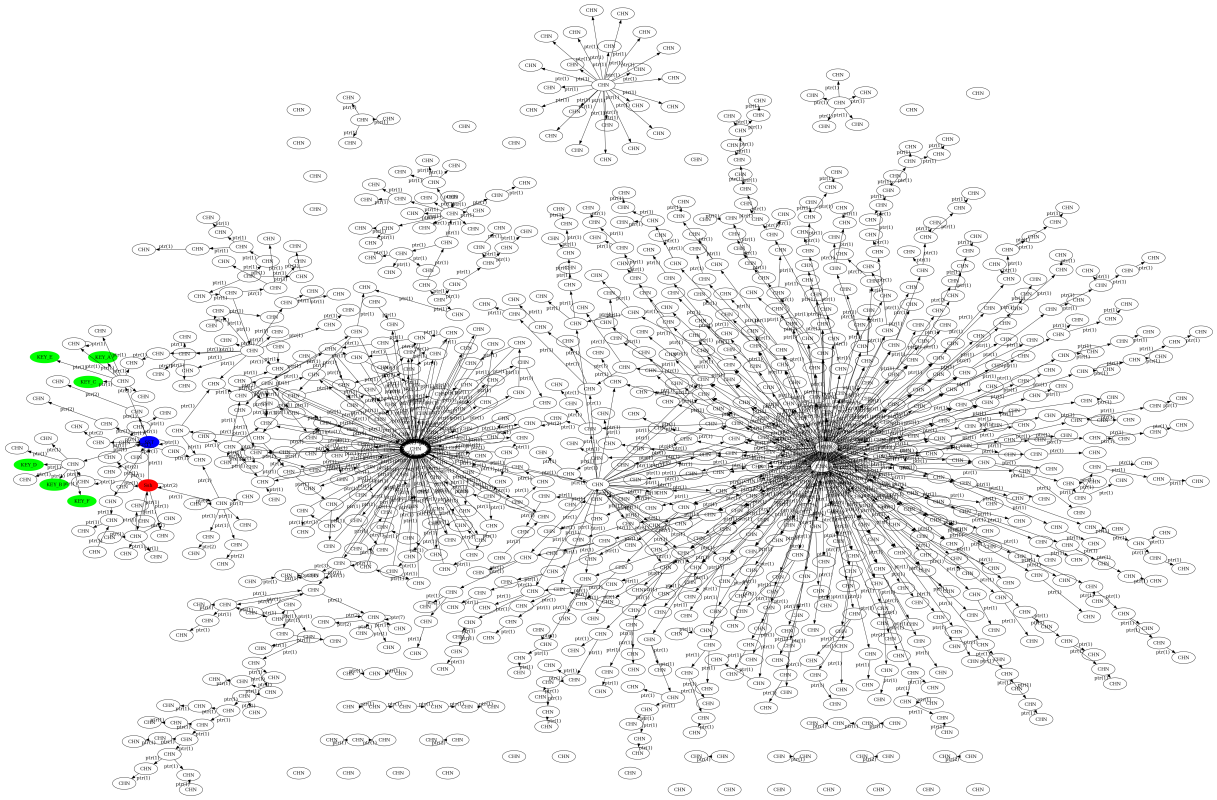


Figure 4: Visualization of a chunk memory graph generated from *Validation/Validation/basic/V_7_-8_P1/24/8708-1643979488-heap.raw*.

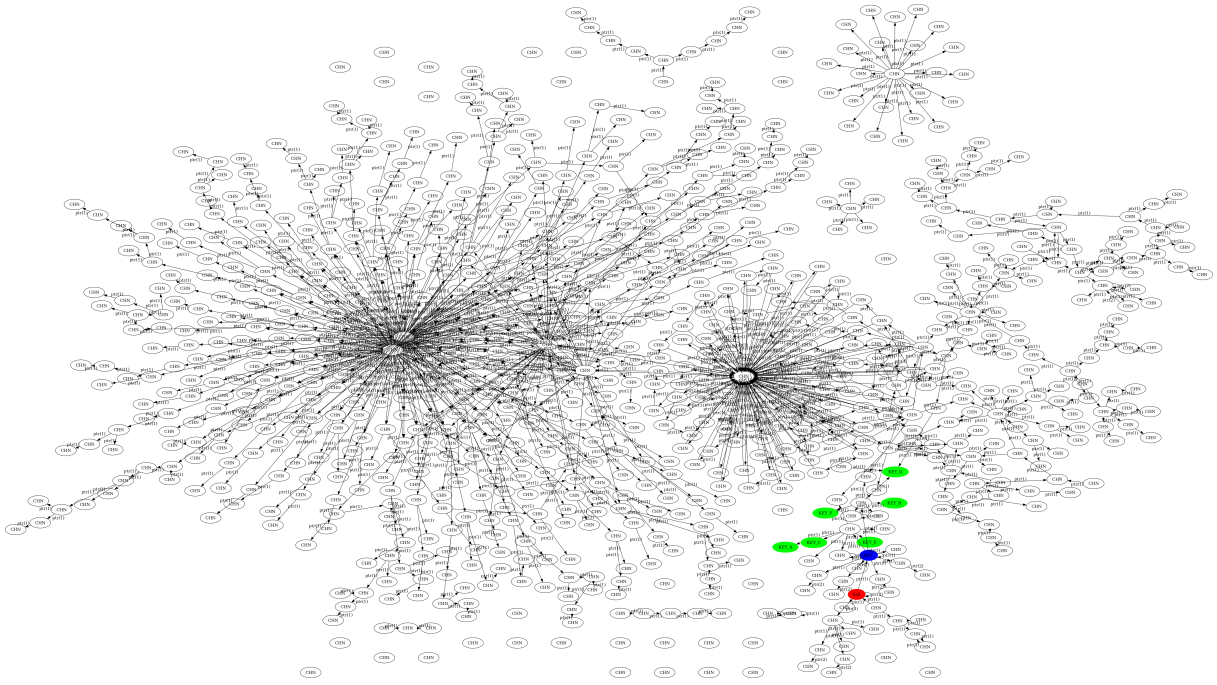


Figure 5: Visualization of a chunk memory graph generated from *Training/Training/basic/V_6_8_-P1/24/28621-1643890740-heap.raw*.

Acronyms

AI Artificial Intelligence. 18

DEL Directed Edge-labelled Graphs. 16

DL Deep Learning. 96, 118

ESS Estimated Security Strength. 8

FE Feature Evaluation. 96

GCN Graph Convolutional Networks. 2, 96, 104, 110, 116, 117

GNN Graph Neural Network. 32, 33

GRU Gated Recurrent Units. 31

KG Knowledge Graph. 12, 16, 17, 74, 116

KNN K-Nearest Neighbors. 20

LDA Linear Discriminant Analysis. 20

LLM Large Language Model. 24

LSB Least Significant Bit. 61, 64, 69

ML Machine Learning. 2, 18, 20, 28, 29, 40, 41, 49, 58–60, 76, 87, 96, 117, 118

NLP Natural Language Processing. 22

OWL Web Ontology Language. 17

PCA Principal Component Analysis. 20

RDF Resource Description Framework. 16, 17, 74

RNN Recurrent Neural Networks. 30

SMOTE Synthetic Minority Over-sampling Technique. 37

SSH Secure Shell Protocol. ii, 5, 8, 116

SVM Support Vector Machine. 20

t-SNE t-distributed Stochastic Neighbor Embedding. 20

VMI Virtual Machine Introspection. 36

Glossary

CHN Chunk Header 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 malloc-allocated memory chunk.. 75

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.. 75

memory graph A memory graph, or *memgraph* is a graph representation of a memory dump. This graph can be a graph of blocks, where each node in the graph corresponds to a block of 8 bytes in the heap dump and each edge corresponds to a pointer from one block to another, or describes which blocks are part of a chunk whose root note is a Chunk Header Node. It can also be a graph of chunks (only CHNs), where each node in the graph corresponds to a chunk in heap dump and each edge corresponds to a pointer from one object to another.. 77

PN Pointer Node. This is a node whose bytes have been identified as a pointer.. 75

VN Value Node. These are all blocks that have not been identified. It is the default node type.. 75

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] Clement Lahoche. *Structure embeddings for OpenSSH heap dump analysis*. en. Available online. 2023. url: https://github.com/passau-masterarbeit-2023/masterarbeit-report-clement (visited on 11/01/2023).

[3] Matthias Fey and Jan Eric Lenssen. *Fast Graph Representation Learning with PyTorch Geometric*. 2019. arXiv: 1903.02428 [cs.LG].

[4] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot*. Accessed: 2023-10-27. Jan. 2015. url: https://graphviz.org/pdf/dotguide.pdf.

[5] 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).

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

[7] 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.

[8] 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).

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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/.

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

[15] 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.

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

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

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

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

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

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

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

[23] 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.

[24] 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.

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

[26] F. Pedregosa et al. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011). Software available at https://scikit-learn.org/stable/index.html, pp. 2825–2830.

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

[28] 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.

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

[30] 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).

[31] 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).

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] Ralph C. Merkle. „Secure Communications over Insecure Channels“. In: *Commun. ACM* 21.4 (Apr. 1978), pp. 294–299. issn: 0001-0782. doi: 10.1145/359460.359473. url: https://doi.org/10.1145/359460.359473.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

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

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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).

[59] 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).

[60] 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).

[61] 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.

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

[63] 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).

[64] 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.

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

[66] 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.

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

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

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

[74] 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.

[75] 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.

[76] 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.

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

[78] 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.

[79] 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.

[80] 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.

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

[82] 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.

[83] 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.

[84] 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.

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

[86] 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.

[87] 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.

[88] 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.

[89] 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/.

Additional bibliography

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

[91] 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.

[92] 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.

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

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

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

[96] 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/.

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

[98] 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 Masterarbreit 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 Masterarbreit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, November 6, 2023

_____

Rascoussier, Florian Guillaume Pierre