Masterarbeits

Structure embeddings for OpenSSH heap dump analysis

A report by

Lahoche, Clément Claude Martial

Prüfer

Prof. Dr. Michael Granitzer

Christofer Fellicious

Prof. Dr. Pierre-Edouard Portier

October 4, 2023

Abstract

Acknowledgements

Contents

List of Figures

List of Tables

Algorithms and program code

1 Introduction

Digital forensics is a linchpin in cybersecurity, enabling the extraction of vital evidence from devices like PCs. This evidence is key for detecting malware and tracing intruder activities. Analyzing a device's main memory is a go-to technique in this field. The fusion of machine learning promises to amplify and streamline these analyses.

With the rising need for encrypted communication, Secure Shell (SSH) protocols are now commonplace. However, these security-focused channels can inadvertently shield malicious actions, posing challenges to standard investigative approaches. Cutting-edge research offers solutions. The work in *SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump* [9] highlights how machine learning can boost the extraction of session keys from OpenSSH memory images. In a complementary vein, „SSHkex: Leveraging virtual machine introspection for extracting SSH keys and decrypting SSH network traffic“ [31] showcases the power of Virtual Machine Introspection (VMI) for direct SSH key extraction.

Inspired by *SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump*, this thesis zeroes in on a central challenge: data embedding. While previous studies set the stage for key extraction, the data embedding technique, especially windowing, can be optimized. The design of data embeddings is pivotal for machine learning efficacy, especially in nuanced tasks like memory analysis. This research introduces fresh embedding strategies, aiming to refine extraction and unearth deeper memory snapshot patterns. Merging graph embeddings with advanced machine learning, the goal is to craft a sophisticated toolkit for OpenSSH heap dump studies, bridging digital forensics and machine learning.

2 Research Questions

- What are the most effective techniques for embedding byte sequences, especially when aiming to extract structures containing SSH keys for machine learning purposes?
- Do the embeddings designed show noticeable differences based on the various applications of OpenSSH, considering the wide range of SSH key sizes and the complex operations of OpenSSH?
- How can we ensure the consistency and stability of these embeddings across the wide variety of OpenSSH versions and usages?

3 Structure of the Thesis

Explain the structure of the thesis.

4 Related work

The embedding of memory heap dumps for the detection of SSH keys is a niche yet crucial area of research, especially in the context of cybersecurity and digital forensics. This section reviews two seminal papers that have significantly influenced our work: *SSHKex*, which delves into Virtual Machine Introspection (VMI) and memory forensics, and *SmartKex*, which employs machine learning techniques for SSH key detection.

4.1 Virtual Machine Introspection and Memory Forensics: SSHKex

SSHKex is an initiative that delves into the intricacies of analyzing encrypted SSH traffic. By harnessing the capabilities of VMI, Sentanoe and Reiser spearheaded this project to discreetly extract SSH keys and decrypt SSH network communications, ensuring the preservation of evidence [31].

The methodology adopted by **SSHKex** seamlessly integrates conventional network traffic monitoring with dynamic SSH session key retrieval. A pivotal assumption is the familiarity with the SSH server's implementation, which is vital for the extraction process. Tools such as LibVMI and Volatility, under the VMI umbrella, are employed to provide an unaltered perspective of the guest VM's state, enabling the efficient pinpointing of SSH session keys within a Linux system's primary memory.

Outlined below is the **SSHKex** key extraction procedure:

1. **Data Structure Insights**: The technique capitalizes on in-depth understanding of the data structures housing the keys. Debugging symbols, tailored to the SSH version on the target, offer crucial offset values, aiding in key material extraction. Key structures encompass `struct ssh`, `struct session_state`, `struct newkeys`, and `struct sshenc`, which collectively house details like IP addresses, session statuses, and encryption keys.

2. **OpenSSH Function Tracing**: This step involves tracing functions to accurately locate data structures and timely key extraction. Emphasis is on `kex_derive_keys` (for key generation initiation) and `do_authentication2` (triggering user authentication).

3. **Breakpoint Implementation**: For debugging purposes, software breakpoints are strategically embedded in the program's execution. Using VMI, SSHKex introduces these breakpoints at the starting points of the two pivotal functions mentioned above.

4. **Extraction of Keys**: The `kex_derive_keys` function's invocation prompts SSHKex to initially capture the `ssh struct`'s address. The subsequent call to the `do_authentication2` function facilitates the extraction of actual keys, adhering to recognized structures.

5. **Key Classification**: OpenSSH designates distinct indices in the `newkeys` structure for client-to-server and vice versa keys. SSHKex's extraction is based on these specific indices.

6. **Managing Multiple Sessions**: OpenSSH handles numerous SSH sessions by initiating child processes. SSHKex broadens its extraction approach to each child process, identifying them via their distinct process IDs.

A standout feature of **SSHKex** is its commitment to discretion, conservation, and maintaining evidence authenticity. The methodology is crafted to minimize intrusiveness, ensuring no alterations to the scrutinized system. This is paramount in forensic scenarios where evidence sanctity is of utmost importance.

4.2 Machine Learning for SSH Key Detection: SmartKex

SmartKex builds upon the foundation of extracting SSH keys from heap memory dumps, aiming to streamline and automate the process. The project stands out by integrating machine learning techniques, enhancing the efficiency and precision of key extraction. This contrasts with the more complex SSHKex approach, which necessitates in-depth SSH knowledge and breakpoint injections.

SmartKex proposes two key extraction methods:

- *Brute-Force Baseline Method*: A conventional method that sifts through heap memory, identifying potential keys based on recognized patterns.
- *Machine Learning-Assisted Method*: Utilizes a Random Forest algorithm, trained on an imbalanced dataset balanced using SMOTE. While this method offers high precision and recall, it's probabilistic, making it less exact than the brute-force approach.

4.2.1 Baseline Brute-Force Method

The brute-force approach of **SmartKex** encompasses the following steps [9]:

1. *Heap Dump Creation*: Binary files of the OpenSSH server process are generated (methodology unspecified in SmartKex) and are presumed to be based on a linux-x86_64 architecture.
2. *Data Trimming*: The method trims irrelevant memory pages based on Hamming distance to reduce heap size.
3. *Key Search*: The algorithm scans the heap, considering a 128-byte length as a potential key, iterating until the heap's end.
4. *Decryption Trials*: Each potential key undergoes decryption attempts on network packets. Failed attempts lead to the next potential key.

Despite its exactness, the brute-force method is resource-intensive and is less efficient when keys are towards the heap dump's end.

4.2.2 Machine Learning-Assisted Method

SmartKex's innovation lies in its machine learning methodology, which, while sacrificing exactness, offers speed and accuracy. The method also reduces the heap to under 2% of its original size. The steps include:

1. *Heap Dump Inputs*: As with the brute-force method, binary files from OpenSSH are the primary inputs.
2. *Data Preprocessing*: The heap dump is reshaped into an N $\times$ 8 matrix. High entropy sections, potential encryption keys, are flagged using logical operations on byte differences.
3. *Model Training*: A Random Forest algorithm is trained on 128-byte segments of the processed heap. Given the dataset's imbalance, a stacked classifier approach is employed.
4. *Key Detection*: Predictions on potential key-containing slices are made using the model, followed by a brute-force extraction.

SmartKex not only outperforms the brute-force method in speed but also excels in accuracy. Its applications span across cybersecurity and memory forensics. The adaptability of its machine learning methodology makes it a valuable asset for both researchers and professionals. The project's open-source nature further enhances its accessibility, with the code available on GitHub.

4.2.3 Our Contribution

Building upon the foundational work of *SSHKex* and *SmartKex*, our research aims to further the field by [Your Contribution Here: e.g., "developing a hybrid approach that combines the strengths of both VMI and machine learning for more accurate and efficient SSH key detection in memory heap dumps."]

5 Background

In the complex world of cybersecurity and digital forensics, innovative approaches are crucial for revealing hidden or encrypted information. OpenSSH stands out as a key instrument for ensuring secure communication. The memory snapshots, or heap dumps, of OpenSSH are treasure troves of data. Through graph generation from these dumps, we can uncover the detailed connections between data structures, identified by their malloc headers, and their associated pointers.

This research delves deep into the smart embedding of these connections, aiming to use machine learning classifiers to identify structures that contain OpenSSH keys. The journey is not just about representing data through graphs but also about understanding the raw sequences of bytes in the heap dump. Classical techniques like Shannon entropy, Byte Frequency Distribution (BFD), and bigram frequencies provide foundational knowledge. However, the rapidly evolving domain of deep learning opens up a plethora of avenues. Models such as Recurrent Neural Networks (RNN) [20] (Long Short-Term Memory (LSTM)[13] and Gated Recurrent Units (GRU)[4]) and sequence-to-sequence learning [33] offer unique perspectives on raw byte embedding. Furthermore, the efficacy of convolutional approaches (CNN), both standalone[21] and in conjunction with recurrent networks, for sequence modeling is well-documented [1]. Notably, the application of neural networks in file fragment classification, especially with lossless representations, has shown promising results [11].Finally, we will dive into transformers[34] and autoencoders.

The aim of this background section is to provide a comprehensive overview of graph creation from heap dumps, techniques for raw byte embedding, and their role in identifying OpenSSH key structures. By merging age-old techniques with modern approaches, we strive to highlight the most effective methods for analyzing OpenSSH heap dump.

5.1 Graph Generation from Heap Dumps

5.1.1 Secure Shell (SSH)

„The Secure Shell (SSH) is designed to enable encrypted communication across potentially unsecured networks, ensuring the confidentiality of data during transmission. Each SSH session utilizes a specific set of session keys, encompassing six distinct keys:

- **Key A:** Client-to-server initialization vector (IV)
- **Key B:** Server-to-client initialization vector (IV)
- **Key C:** Client-to-server encryption key (EK)
- **Key D:** Server-to-client encryption key (EK)

- **Key E:** Client-to-server integrity key
- **Key F:** Server-to-client integrity key

To decrypt the encrypted traffic within an SSH session, knowledge of the IV and EK pair (either Key A with Key C or Key B with Key D) is essential, assuming the presence of passive network monitoring tools. OpenSSH, a prevalent implementation of SSH, is the primary subject of this research, covering versions from V6_0P1 to V8_8P1. OpenSSH incorporates various encryption methodologies, including Advanced Encryption Standard (AES) Cipher Block Chaining (CBC), AES Counter (AES-CTR), and ChaCha20, with IV and EK key lengths varying between 12 and 64 bytes.“

This information is derived from the paper titled *SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump* [9].

5.1.2 heap dumps of OpenSSH

„Heap memory, distinct from local stack memory, is a dynamic memory allocation mechanism. While local stack memory is responsible for storing and deallocating local variables during function calls, heap memory requires explicit memory allocation and deallocation. This is achieved using operators such as `new` in Java and C++, or `malloc/calloc` in C.

OpenSSH, which is primarily written in C, employs `calloc` for memory block allocation. These blocks are designated to store session-related data, including the cryptographic keys. By leveraging this knowledge, one can deduce that if the heap of an active OpenSSH process is dumped at an opportune moment (for instance, during an ongoing SSH session), the resulting heap dump will encompass the SSH session keys.“

This information is also derived from the paper titled *SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump* [9].

5.1.3 Dataset

„We use SSHKex[31] as the primary method to extract the SSH keys from the main memory. In addition, we add two features to SSHKex: automatically dump OpenSSH's heap and add support for SSH client monitoring.

For this paper, we are using four SSH scenarios: the client connects to the server and exits immediately, port-forward, secure copy, and SSH shared connection. Two file formats, JSON and RAW, are utilized to store the generated logs. The JSON log file encompasses meta-information, including the

encryption name, the virtual memory address of a key, and the key's value in hexadecimal representation (as depicted in Figure 5.1). Conversely, the binary file captures the heap dump of the OpenSSH process (illustrated in Figure 5.2 using the `xxd` command).

```
(base) [onyr@kenzael phdtrack_data]$ cat ./Training/Training/scp
/V_7_8_P1/16/1010-1644391327.json | json_pp
{
   "ENCRYPTION_KEY_1_NAME" : "aes128-ctr",
   "ENCRYPTION_KEY_1_NAME_ADDR" : "558b967f7620",
   "ENCRYPTION_KEY_2_NAME" : "aes128-ctr",
   "ENCRYPTION_KEY_2_NAME_ADDR" : "558b967fb160",
   "HEAP_START" : "558b967e9000",
   "KEY_A" : "119bd34f49d27bbbc0f9af400d4edc39",
   "KEY_A_ADDR" : "558b967fefe0",
   "KEY_A_LEN" : "16",
   "KEY_A_REAL_LEN" : "16",
   "KEY_B" : "8a77835eb2007a46a776ae0c183253b9",
   "KEY_B_ADDR" : "558b967f5ce0",
   "KEY_B_LEN" : "16",
   "KEY_B_REAL_LEN" : "16",
   "KEY_C" : "528f6dbd2907b3b4cfbd02fb32b852e7",
   "KEY_C_ADDR" : "558b967f51f0",
   "KEY_C_LEN" : "16",
   "KEY_C_REAL_LEN" : "16",
   "KEY_D" : "427f04149eed7029f031e58f3fde9844",
   "KEY_D_ADDR" : "558b967fb180",
   "KEY_D_LEN" : "16",
   "KEY_D_REAL_LEN" : "16",
   "KEY_E" : "17b6c799b5639ce5ea60c7f67cf6177f",
   "KEY_E_ADDR" : "558b967ff070",
   "KEY_E_LEN" : "16",
   "KEY_E_REAL_LEN" : "16",
   "KEY_F" : "fb75f5776184794ca92624ec6a36fd62",
   "KEY_F_ADDR" : "558b967f3d90",
   "KEY_F_LEN" : "16",
   "KEY_F_REAL_LEN" : "16",
   "NEWKEYS_1_ADDR" : "558b96800fd0",
   "NEWKEYS_2_ADDR" : "558b967fef10",
   "SESSION_STATE_ADDR" : "558b967f7f30",
   "SSH_PID" : "1010",
   "SSH_STRUCT_ADDR" : "558b967f6c20",
   "enc_KEY_OFFSET" : "0",
   "iv_ENCRYPTION_KEY_OFFSET" : "40",
   "iv_len_ENCRYPTION_KEY_OFFSET" : "24",
   "key_ENCRYPTION_KEY_OFFSET" : "32",
   "key_len_ENCRYPTION_KEY_OFFSET" : "20",
   "mac_KEY_OFFSET" : "48",
   "name_ENCRYPTION_KEY_OFFSET" : "0",
   "newkeys_OFFSET" : "344",
   "session_state_OFFSET" : "0"
}
```

Figure 5.1: Json exemple

```
000159d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000159e0: 0000 0008 0000 0000 0080 0000 0000 0000  ................
000159f0: 0000 0000 0000 0000 0100 0000 0000 0000  ................
00015a00: 0000 0000 0000 0000 2100 0000 0000 0000  ........!.......
00015a10: 8d08 ff65 b3bf cd8b 91ca 995a d5b7 64af  ...e.......Z..d.
00015a20: 0000 0000 0000 0000 2100 0000 0000 0000  ........!.......
00015a30: 756d 6163 2d36 342d 6574 6d40 6f70 656e  umac-64-etm@open
00015a40: 7373 682e 636f 6d00 5100 0000 0000 0000  ssh.com.Q.......
00015a50: b0d4 36d2 a655 0000 b0d4 36d2 a655 0000  ..6..U....6..U..
```

Figure 5.2: Xxd exemple

The dataset is structured into two primary directories: `training` and `validation`. Each of these directories is further segmented into subdirectories reflecting the specific scenario, such as OpenSSH, port-forwarding, or secure copy (SCP).

Subdirectories under OpenSSH or SCP are categorized based on the software version responsible for the memory dump. These directories are further organized by the software version that generated the memory dump. The heaps are then classified based on their key lengths, with each key length possessing its dedicated directory beneath the version directory. These version-specific directories are further divided based on the different key lengths present in a heap.

Accompanying every raw memory dump is a JSON file, distinguished by the same alphanumeric sequence, barring the "-heap" suffix. This JSON file encapsulates various encryption keys and additional metadata, such as the process ID and the offset of the heap. Consequently, the dataset's utility is not confined to extracting session keys but also extends to identifying crucial data structures harboring sensitive information. The dataset, along with the associated code and tools, is open-sourced. The dataset is accessible via a Zenodo repository[1]. The code can be found in a public GitHub repository[2]."

This data is the same as the data used in the paper titled *SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump* [9]. The dataset is still accessible [8].

5.1.4 Entropy's Role in SSH Key Identification

Encryption keys[9] inherently consist of predominantly random byte sequences. This characteristic stems from the foundational principle of ensuring security through transparency, which guarantees their high entropy. The paper explores the nuances of pinpointing these keys in memory dumps, underscoring the significance of entropy in this endeavor. This particularity can be used to identify the keys in the memory dump.

5.1.5 Definitions : Structures, Pointers, and the role of malloc headers

5.1.5.1 Pointers

Through the use of the regular expressions (REGEX) `"[0-9a-f]{12}0{4}"`, we identified potential pointers within the dump. This heuristic approach acts as a sieve, filtering the extensive data to spotlight possible pointer candidates. Nonetheless, it's crucial to understand that while many pointers might be correctly pinpointed, some detected sequences may not be authentic pointers.

[1] `https://zenodo.org/record/6537904`
[2] `https://github.com/smartvmi/Smart-and-Naive-SSH-Key-Extraction`

5.1.5.2 Malloc header

Given that OpenSSH is developed in C, it's anticipated that the heap dump files will contain C data structures. In C, memory allocation is typically achieved using the `malloc` function. This function, when invoked, requires the size of the data structure to be allocated and subsequently returns a pointer to the allocated memory space. An examination of the `malloc` code[3] reveals:

```
1                   /* Additional space to account for the size of the
                       allocated block. */
2                   size += MALLOC_HEADER_SIZE;
```

Code 5.1: uclibc's Malloc Implementation

A call to `malloc` typically results in the allocation of a memory block that's the sum of the data structure's size and an additional 8 bytes. This extra allocation is attributed to the metadata that `malloc` stores about the memory block, which is housed in the bytes allocated beyond the data structure's size. Consequently, heap dump files are likely to contain 8-byte aligned blocks, which, while not pointers, emerge from a `malloc` invocation. Identifying these *malloc headers* is instrumental in detecting potential data structures within the heap dump.

While the exact `malloc` implementation used for the OpenSSH programs that generated the dataset remains uncertain, it's plausible that it mirrors the one in uclibc—a streamlined C library prevalent in embedded systems.

On a `x86_64` Linux architecture, the `malloc` function typically employs a block (or chunk) header to retain metadata about each allocated segment. Positioned right before the user-returned memory block, this header's layout might differ based on the C library in use (e.g., glibc, musl). However, it generally encompasses:

- **Block Size**: The allocated block's size, typically measured in bytes. This often accounts for the header's size and might be aligned to 8 or 16 bytes.
- **Flags**: Various indicators that reflect the block's status—whether it's free or allocated, or if the preceding block is free or allocated. These flags are frequently stored in the size field's least significant bits, leveraging the alignment-induced zeroed least significant bits.

Given the system's endianness, the heap dump file is expected to present the malloc header in a little-endian format.

[3]The dataset was generated on a `x86_64` architecture. For illustrative purposes, we've considered the `malloc` implementation from uClibc: `https://git.busybox.net/uClibc/tree/libc/stdlib/malloc/malloc.c`

5.2 Traditional Statistical Embedding

Within the domain of machine learning, how data is represented significantly impacts the performance of models. Even though traditional statistical embedding techniques have been around before many contemporary methods, they continue to be vital in readying data for machine learning endeavors. Rooted in statistical foundations, these techniques provide a methodical approach to transform raw data into concise and meaningful forms. In this subsection, we'll delve into the nuances of entropy and its role in byte sequence embedding, Byte Frequency Distribution (BFD), and also highlight other classical statistical embedding methods pivotal in data representation for machine learning.

5.2.1 Entropy and its application in byte sequence embedding

Entropy, a fundamental concept in information theory, quantifies the amount of uncertainty or randomness associated with a set of data. Introduced by Claude Shannon in his groundbreaking work [32], entropy serves as a measure of the average information content one can expect to gain from observing a random variable's value.

Mathematically, the entropy $H(X)$ of a discrete random variable X with possible values $\{x_1, x_2, \ldots, x_n\}$ and probability mass function $P(X)$ is given by:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i) \tag{5.1}$$

Within the scope of identifying SSH keys, the significance of entropy cannot be understated. Byte sequences exhibiting high entropy typically reflect a multifaceted and varied informational content, traits that are synonymous with encryption keys, especially those in SSH. Sequences with pronounced entropy are often prime contenders for SSH keys due to their inherent randomness and lack of predictability, mirroring the attributes of robust security keys.

Fundamentally, entropy acts as a quantitative tool to evaluate the depth of information within data. When applied to SSH, it suggests that data sequences with elevated entropy levels have a heightened probability of correlating with secure keys. This positions entropy as an essential instrument for pinpointing and authenticating SSH keys.

5.2.2 Byte Frequency Distribution (BFD)

In the complex world of raw byte embedding, Byte Frequency Distribution (BFD) and n-gram embedding stand out as essential methods, each bringing unique benefits to data representation. BFD zeroes in on the distribution of individual byte values in a raw byte sequence. Analyzing these distributions allows for the identification of patterns that reflect the inherent nature of the data. This

embedding technique becomes particularly relevant when assessing the randomness or structure of byte sequences, such as when detecting encrypted data or pinpointing specific file signatures.

On the other hand, n-gram embedding dives deeper into raw byte sequences. Instead of focusing solely on individual bytes, it captures patterns formed by sequences of 'n' consecutive bytes. This approach garners a wider range of contextual information from the raw byte data. For example, a trigram (3-gram) examines patterns formed by three sequential bytes, providing a richer representation than single byte values. Yet, a challenge with n-gram embedding is the potential for the output vector size to grow exponentially as 'n' increases, posing computational and storage issues, especially in real-time scenarios.

In the realm of raw byte embedding, both BFD and n-gram techniques offer invaluable perspectives. While BFD establishes a base representation centered on individual byte frequencies, n-gram embedding enhances it by spotlighting the complex relationships and patterns among consecutive bytes. Together, they form a robust arsenal for representing and analyzing raw byte data in a variety of applications.

5.2.3 Other traditional statistical embedding techniques

Mean Byte Value The Mean Byte Value represents the average value of all bytes in a given sequence. It provides an insight into the central tendency of the byte values in the sequence. Mathematically, for a byte sequence B of length n:

$$\text{Mean Byte Value} = \frac{1}{n}\sum_{i=1}^{n} B_i \tag{5.2}$$

Mean Absolute Deviation (MAD) MAD measures the average distance of each byte value from the mean, providing a sense of the dispersion or spread of the byte values around the mean. It is given by:

$$\text{MAD} = \frac{1}{n}\sum_{i=1}^{n} |B_i - \text{Mean Byte Value}| \tag{5.3}$$

Standard Deviation Standard Deviation quantifies the amount of variation or dispersion in the byte sequence. A higher value indicates greater variability in the byte values. It is defined as:

$$\text{Standard Deviation} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(B_i - \text{Mean Byte Value})^2} \tag{5.4}$$

Skewness Skewness[37] measures the asymmetry of the distribution of byte values around the mean. A positive value indicates a distribution that is skewed to the right, while a negative value indicates a distribution skewed to the left. It provides insights into the shape of the distribution of byte values.

The Fisher's skewness[3] is :

$$\text{Skewness} = \frac{n}{(n-1)(n-2)} \sum_{i=1}^{n} \left(\frac{B_i - \text{Mean Byte Value}}{\text{Standard Deviation}} \right)^3 \tag{5.5}$$

Kurtosis Kurtosis[37] measures the "tailedness" of the distribution of byte values. A higher kurtosis value indicates a distribution with heavier tails, while a lower value indicates lighter tails. It provides insights into the extremities of the distribution. The Fisher's kurtosis[3] is :

$$\text{Kurtosis} = \frac{n(n+1)}{(n-1)(n-2)(n-3)} \sum_{i=1}^{n} \left(\frac{B_i - \text{Mean Byte Value}}{\text{Standard Deviation}} \right)^4 - \frac{3(n-1)^2}{(n-2)(n-3)} \tag{5.6}$$

n-gram on Bits When applying n-gram techniques to bits instead of bytes, we focus on sequences of 'n' consecutive bits. For example, a 2-gram on bits would consider patterns formed by two consecutive bits, resulting in four possible combinations: 00, 01, 10, and 11. This approach significantly reduces the size of the output vector compared to byte-based n-grams. By focusing on bits, we can capture more granular patterns in the data while benefiting from a more compact representation, which is computationally efficient and requires less storage.

5.3 Deep Learning Models for Raw Byte Embedding

In the area of data representation, deep learning is great for understanding raw byte sequences. Just like these models are good at understanding text, they're also good at understanding raw bytes. They can learn and show sequences on their own, which is really helpful for both text and raw bytes. In this section, we'll look at different deep learning models and how they work with raw byte embedding.

We'll start with Recurrent Neural Networks (RNN). Just like they're good with word sequences in text, Recurrent Neural Networks (RNN) are also good with raw byte sequences. Then, we'll look at Convolutional Neural Networks (CNN), which can find patterns in raw bytes, just like they find patterns in text. After that, we'll talk about Autoencoders, which can learn in a special way. To finish this section, we'll discuss Transformers. They're good at understanding data over a long time, similar to how they understand text.

5.3.1 RNNs : Understanding sequence data

Recurrent Neural Networks (RNN) are great tools for text classification. They're good at understanding the deeper meanings in text. Unlike older models that use hand-made features, RNN can learn and show sequences on their own. This makes them really useful for tasks that deal with sequences. When we think about embedding raw bytes, RNN's skill in understanding sequences is similar to how they handle word sequences in text. Here is a list of different RNN models and their advantages and disadvantages.

Recurrent Convolutional Neural Network (RCNN) for Text Classification[20]: The RCNN model, as discussed in the paper by Lai et al., is designed specifically for text classification. Unlike traditional models, RCNN do not rely on handcrafted features. Instead, they employ a recurrent structure to capture contextual information about words. This approach is believed to introduce considerably less noise compared to traditional window-based neural networks. The model's bidirectional structure ensures that both preceding and succeeding contexts of a word are considered, enhancing its understanding of the word's semantics.

- **Advantages:**
  - No need for handcrafted features.
  - Captures richer contextual information.
  - less noisy.
- **Disadvantages:**
  - Complexity due to bidirectional structure.
  - Might require more computational resources.

;

Long Short-Term Memory (LSTM)[13]: The LSTM, introduced by Hochreiter and Schmidhuber, is a specialized form of RNN designed to combat the vanishing gradient problem inherent in traditional RNN. The vanishing gradient problem arises when gradients of the loss function, which are used to update the network's weights, become too small for effective learning. This typically happens in deep networks or when processing long sequences, causing the earlier layers or time steps to receive minimal updates. As a result, traditional RNN struggle to learn long-term dependencies in the data.

LSTM address this issue with their unique cell state and gating mechanisms. The cell state acts as a "conveyor belt" that can carry information across long sequences with minimal changes, ensuring that long-term dependencies are captured. The gating mechanisms, namely the input, forget, and output gates, regulate the flow of information into, out of, and within the cell. This design allows LSTMs to selectively remember or forget information, making them adept at learning and retaining long-term dependencies in sequences.

- **Advantages:**
  - Efficiently learns long-term dependencies; overcomes the vanishing gradient problem inherent in traditional RNN.
  - Often achieves faster and more stable learning.
- **Disadvantages:**
  - More complex architecture compared to basic RNN and even GRU.
  - Can be computationally intensive due to the multiple gating mechanisms.

Gated Recurrent Units (GRU)[4]: GRU are a variant of RNN that aim to capture long-term dependencies without the complexity of LSTM. They use a gating mechanism to control the flow of information, making them efficient in sequence modeling tasks.

- **Advantages:**
  - Simplified structure compared to LSTM.
  - Efficient in capturing long-term dependencies.
  - Sometimes outperforms LSTM.
- **Disadvantages:**
  - Still more complex than traditional RNN.
  - Might not always outperform LSTM in all tasks.

To sum it up, RNN are good at understanding sequences and context. This makes them a good choice for embedding raw bytes. Just like they understand words based on the words around them, RNN can find patterns in raw byte sequences, giving us a better understanding of the data.

5.3.2 CNNs : Pattern detection in raw bytes

Convolutional Neural Networks (CNN)[21] are a specialized category of deep learning models adept at identifying patterns. Originally designed for visual data, their prowess extends to tasks like image and document recognition. Drawing inspiration from the human visual cortex's biological processes, CNN are architected to autonomously and adaptively discern spatial feature hierarchies from inputs. This becomes particularly relevant when considering raw byte embedding, where the goal is to detect patterns in sequences of bytes. The CNN architecture boasts convolutional layers that perform operations on input data to capture localized patterns, and pooling layers that condense spatial dimensions while preserving crucial information. This layered approach enables CNN to detect intricate patterns by progressively building on simpler foundational patterns. When applied to byte sequences or document recognition, CNN excel, showcasing remarkable efficacy, especially in tasks like identifying patterns within raw byte sequences or recognizing handwritten content.

When tailored to CNN, the Sequence-to-Sequence (Seq2Seq)[10] approach emerges as a potent tool for transforming raw byte sequences into meaningful embeddings. The encoder segment of the Seq2Seq model is central to this transformation. It delves into the byte sequence, discerning intricate patterns and nuances, and distills this rich information into a concise context vector or embedding. This condensed representation captures the core essence of the byte sequence, positioning it as a valuable input for subsequent tasks, such as classification models.

At the heart of the encoder lie the convolutional layers, skilled in pinpointing specific patterns within the byte sequence. Whether it's unique byte combinations or indicative n-grams, these layers are

primed to detect them. As they traverse the raw byte sequence, they employ specialized filters, honed to recognize these specific patterns. As the data flows through the encoder's layers, these identified patterns are synthesized and refined, culminating in a comprehensive embedding of the sequence.

Here are two Sequence-to-Sequence (Seq2Seq) models using CNN :

- **Autoencoders:** These neural network architectures[12] are designed for data compression and reconstruction. The encoder part compresses the input data into a compact representation, while the decoder reconstructs the original data from this representation. In the context of raw byte sequences, the encoder can be used to generate embeddings that capture the essential patterns and structures of the data.
- **Transformers :** Transformers[34] utilize self-attention mechanisms to weigh the significance of different parts of the input data. This allows them to capture long-range dependencies and relationships in the data. When applied to raw byte sequences, transformers can generate embeddings that consider both local and global patterns, making them particularly effective for tasks that require understanding the broader context of a sequence.

Yet, a significant challenge with traditional Sequence-to-Sequence (Seq2Seq) models using CNN is their constraint in managing inputs of varying sizes. Constructed with a set input size, they face difficulties when presented with sequences of diverse lengths, like raw byte sequences.

To address this limitation, various techniques have been employed to normalize the size of the input data. One of the most common methods is **padding**, where shorter sequences are filled with predefined placeholder values (often zeros) until they match the length of the longest sequence in the dataset. This ensures that all sequences fed into the model have a uniform length. Another approach is **bucketing**, where sequences of similar lengths are grouped together, minimizing the amount of padding required. Additionally, **truncation** can be used to shorten sequences that exceed a certain length, although this might result in the loss of some information. While these techniques enable CNN-based Sequence-to-Sequence (Seq2Seq) models to handle variable-sized inputs, it's crucial to ensure that the preprocessing steps do not introduce noise or distort the inherent patterns and relationships within the raw byte sequences.

5.4 Graph Embedding Methods

Graph embedding techniques focus on the mapping of nodes and edges within a graph onto vectors within a lower-dimensional space [14]. The primary objective is to retain the structural properties of the graph, such as node connectivity and community structure, in this embedded space. These resulting vectors find application in diverse machine learning tasks, including clustering, classification, and link prediction. For a comprehensive exploration of these techniques, "Knowledge Graphs" offers a detailed overview. Some noteworthy techniques in this domain encompass:

1. **Translational Models:** These graph embedding techniques revolve around translational models that interpret edge labels as transformations from subject nodes to object nodes [14].

    - **TransE:** One of the earliest and straightforward translational models, TransE represents entities as points in a vector space and relations as translations between these points. The core idea is that for a valid triple (h, r, t), the equation $h + r = t$ should hold, with h representing the head entity, r denoting the relation, and t standing for the tail entity. While simple and computationally efficient, this model has limitations in capturing complex relationships [14].
    - **TransH:** Extending TransE, TransH introduces relation-specific hyperplanes. This extension allows the model to capture more intricate relationships by projecting entity embeddings onto these hyperplanes before performing translations [14].
    - **TransR:** Going a step further than TransH, TransR not only introduces relation-specific hyperplanes but also relation-specific translations. This flexibility enables a more versatile representation of relations, accommodating various complexities [14].
    - **Other Developments:** Noteworthy advancements include TransD and MuRP, highlighting the active research in this domain [14].

2. **Tensor Decomposition Models** These models represent entities and relations as vectors or matrices within a lower-dimensional space, grounded in the assumption that the relationship between entities can be expressed through a bilinear function. While they are computationally efficient and capable of capturing complex relationships, they do have limitations in modeling asymmetric and reflexive relations.

    - **RESCAL:** RESCAL employs a bilinear model, where each relation is represented by a full-rank matrix. This approach allows for the representation of asymmetric and reflexive relations but comes at the cost of increased computational complexity [14].
    - **DistMult:** DistMult simplifies RESCAL by assuming that the relation matrices are diagonal. This simplification reduces the number of parameters and computational complexity, making it a more scalable option [14].
    - **ComplEx:** ComplEx extends DistMult by introducing complex-valued embeddings. This addition enables the model to effectively capture asymmetric relations while maintaining computational efficiency [14].

3. **Neural Models** Neural models leverage neural networks to acquire the features of entities and relations, offering a flexible and adaptive approach to graph embeddings.

    - **ConvKB:** ConvKB employs a convolutional neural network to autonomously learn entity and relation features. While technically a translational model, it uses a convolutional layer to capture interactions between entities and relations, providing a flexible and adaptable approach to graph embeddings [25].
    - **RotatE:** RotatE introduces complex rotations in the embedding space to model relations. This neural model offers a more expressive approach to capturing relation semantics [14].

- **SDNE (Structural Deep Network Embedding):** SDNE uses a deep autoencoder to acquire complex and non-linear node embeddings while preserving first-order and second-order proximities. It is particularly effective in capturing intricate patterns and structures within the graph [36].
- **R-GCN (Relational Graph Convolutional Networks):** R-GCNs combine the strengths of Graph Convolutional Networks (GCNs) and traditional embedding methods, capturing both topological and semantic information [30]. A recent study by Degraeve et al. highlights the importance of the message passing paradigm in R-GCN and introduces a variant called Random R-GCN (RR-GCN) [5].
- **ConvE:** ConvE employs convolutional layers to capture local and global interactions between entities and relations, offering a more expressive representation [14].

4. **Language Models** Language models harness pre-trained language models to enrich embeddings with contextual information, tapping into recent advancements in language modeling to grasp the semantics of entities and relations.

   - **BERT for KGE (Knowledge Graph Embedding):** This approach utilizes pre-trained BERT models, capitalizing on the capabilities of language models to enhance embeddings with contextual information [39].
   - **BART KGE:** Bidirectional and Auto-Regressive Transformers (BART) serve as denoising autoencoders suitable for various NLP tasks. This method employs BART to acquire entity and relation embeddings, and the associated paper also compares it with other Large Language Models (LLMs) like GPT-2 [22].
   - **It's worth noting that the field of NLP is continuously evolving, with ongoing developments and emerging approaches.**

5.5 Machine learning

Machine learning, an integral part of artificial intelligence, revolves around designing algorithms and statistical models that allow computers to perform tasks without being directly programmed. Instead of relying on detailed instructions for every task, machine learning techniques empower systems to learn from data and make data-driven decisions. A key method in this field is supervised learning, in which models are trained using data that comes with predefined labels. Here, each piece of data in the training set has an associated known output. The primary goal of supervised learning is to establish a relationship between inputs and outputs, enabling the model to predict or categorize new, unseen data based on this relationship.

A cornerstone in this realm is feature engineering, which involves the meticulous process of selecting and transforming variables to optimize model performance. Another challenge frequently encountered by practitioners is dealing with datasets where some classes are overrepresented, which can skew model predictions. Among the myriad of machine learning models available, certain ones have gained prominence due to their versatility and effectiveness. We will provide an overview of some of these notable models.

5.5.1 Features engineering

Feature engineering[16] is a cornerstone in the realm of machine learning. It involves the artful transformation of the given feature space to optimize the performance of predictive models. The significance of feature engineering cannot be overstated; it serves as a bridge between raw data and the predictive models, ensuring that the models are fed with the most relevant and informative features. Properly engineered features can drastically reduce modeling errors, leading to more accurate and reliable predictions. Here are some of the most common feature engineering techniques:

- **Normalization and Scaling** are preprocessing techniques used to standardize the range of independent features in the data. Many machine learning algorithms, especially those that rely on distance calculations like k-means clustering or support vector machines, are sensitive to the scale of the data. If features are on different scales, one feature might dominate others, leading to suboptimal model performance. Normalization typically scales features so that they have a unit norm, while other scaling methods, such as Min-Max scaling, transform features to lie in a given range, usually [0,1]. Z-score normalization or standard scaling is another method where features are scaled based on their mean and standard deviation. Properly scaled data ensures that each feature contributes equally to the model's decision, leading to more stable and accurate predictions.

- **Interaction Features[15]** refer to the creation of new features by combining two or more existing features, aiming to capture any synergistic effect between them. In many cases, the interaction between variables can provide more information than the individual variables themselves. For instance, while analyzing real estate prices, the individual features 'number of rooms' and 'location' might be informative, but their interaction, 'number of rooms in a specific location', might offer even more predictive power. Interaction features can be created by multiplying, adding, or even dividing original features, and they can help in capturing non-linear relationships in the data, enhancing the model's ability to make accurate predictions.

- **Feature Selection[15]** is a critical process in the machine learning pipeline that focuses on selecting the most relevant features from the original set, aiming to reduce the dimensionality and improve model performance. The primary goal is to eliminate redundant or irrelevant features that don't contribute significantly to the predictive power of the model. This not only helps in reducing the computational cost but also can lead to a more interpretable model. There are various techniques for feature selection, including filter methods (based on statistical measures), wrapper methods (like recursive feature elimination), and embedded methods (where algorithms inherently perform feature selection, such as decision trees). By judiciously selecting features, one can build efficient models that are less prone to overfitting and have better generalization capabilities.

5.5.1.1 Correlation tests

To assess feature quality, various statistical measures come into play, including correlation tests that gauge the strength and direction of relationships between variables. Pearson, Kendall, and Spearman

correlation coefficients are frequently employed to quantify linear or monotonic associations between each feature and the target variable [2]. A high absolute value of these coefficients indicates a robust relationship, aiding in feature selection.

- **Pearson Correlation:** This measures linear relationships between two variables, ranging from -1 to 1. -1 signifies a strong negative linear correlation, 1 suggests a strong positive linear correlation, and 0 implies no linear correlation.
- **Kendall's Tau:** This non-parametric test gauges the strength and direction of a monotonic relationship between two variables.
- **Spearman's Rank:** Also non-parametric, it assesses how well an arbitrary monotonic function can describe the relationship between two variables without making assumptions about frequency distribution.

These techniques are valuable for evaluating relationships between each feature and generating correlation matrices, which, in turn, help identify redundant features. Univariate feature selection techniques allow the evaluation of each feature independently. In Python's scikit-learn library [27], methods like the F-test value and p-value are often used for this purpose.

- **F-test value:** This measures the linear dependency between the feature variable and the target. A higher F-test value suggests a more useful feature.
- **p-value:** It indicates the probability of an F-test value as large as observed arising if the null hypothesis is true. A smaller p-value implies rejecting the null hypothesis, indicating the feature's significance.

In summary, features constitute the foundational elements of any machine learning model. The quality of these features, their processing, and utilization significantly impact the model's performance. Feature engineering is of paramount importance, as properly engineered features can substantially reduce modeling errors, leading to more accurate and reliable predictions. It serves as a crucial link between raw data and predictive models, ensuring that models are fed with the most relevant and informative features.

5.5.1.2 Dimensionnality reduction

Following the aforementioned techniques, another essential facet in the feature engineering landscape is dimensionality reduction. As data grows in complexity, it often encompasses a vast number of features, leading to what is known as a high-dimensional space. While a plethora of features might seem advantageous, it introduces challenges, notably the *curse of dimensionality*[35, 17]. In such high-dimensional realms, data points tend to become increasingly sparse. This sparsity means that the relative distances between data points start to appear uniform, making it arduous for algorithms to discern meaningful patterns. This can lead to models that overfit the training data, capturing noise

rather than the underlying data distribution. Additionally, the computational overhead increases, and deriving intuitive insights from the data becomes a daunting task.

Dimensionality reduction techniques come to the rescue by striving to trim down the number of features while preserving the crux of the information. Techniques like Principal Component Analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE) are employed to transform the data from its original high-dimensional space to a more manageable, lower-dimensional one. This transformation aims to retain the significant patterns and structures inherent in the data. By judiciously reducing the dimensionality, not only can models be trained more efficiently, but they often yield better performance by focusing on the most pertinent features. This streamlined approach mitigates the challenges posed by the curse of dimensionality, ensuring models that are both robust and interpretable.

5.5.2 Imbalanced data

In machine learning, a frequent obstacle is the presence of datasets where one category vastly overshadows others[29]. This imbalance can skew models towards predicting the dominant class, often neglecting the less prevalent but potentially more critical class.

To counteract this, a variety of techniques have been devised:

- **Resampling:** This encompasses both increasing instances of the minority class (oversampling) and decreasing instances of the majority class (undersampling). A notable method for oversampling is the Synthetic Minority Over-sampling Technique (SMOTE), which generates artificial data points in the feature space.
- **Weighted Loss:** This strategy involves assigning greater weights to the minority class during the training phase, ensuring the model gives it due consideration.
- **Ensemble Methods:** Approaches such as bagging and boosting can be tailored to ensure a balanced class representation. For example, in bagging, each sample can be structured to maintain a balanced class ratio.
- **Anomaly Detection:** This method reframes the task from classification to anomaly detection, viewing the minority class as an outlier or anomaly.

Selecting an appropriate strategy hinges on the specific problem and dataset characteristics. It's also crucial to evaluate the model's efficacy using suitable metrics, ensuring it genuinely addresses the imbalance.

5.5.3 Some common models

- **Logistic Regression[26] :** Logistic regression serves as a statistical technique tailored for binary classification tasks. While linear regression is designed to forecast continuous outcomes, logistic

regression focuses on predicting the likelihood of a binary result. It leverages the logistic function to relate multiple independent variables to a binary outcome, ensuring the predicted values fall between 0 and 1. Typically, a 0.5 threshold is used to classify the final outcome. A key strength of logistic regression is its clarity and ease of interpretation, though it might face challenges with complex non-linear data unless further feature adjustments are made.

- **Decision Trees[18] :** Decision trees are machine learning models designed for both classification and regression tasks. They segment data into subsets based on feature values, making decisions at each node. While their hierarchical structure offers easy visualization and interpretation, they can be prone to overfitting. However, strategies like pruning can help in refining the tree and mitigating overfitting.

- **Random Forest[28] :** Random Forest is an ensemble method that creates a 'forest' of decision trees. Each tree is trained on a random subset of the data and makes its own predictions. The Random Forest algorithm then aggregates these predictions to produce a final result. This method is known for its high accuracy, ability to handle large datasets with higher dimensionality, and its capacity to manage missing values.

- **Support Vector Machines (SVM)[38] :** SVM are used for both regression and classification problems. They work by finding the hyperplane that best divides a dataset into classes. SVMs are effective in high-dimensional spaces and are versatile, as different Kernel functions can be specified for the decision function.

- **K-Nearest Neighbors (KNN)[19] :**KNN is a simple, instance-based learning algorithm. To make a prediction for a new data point, the algorithm finds the 'k' training examples that are closest to the point and returns the most common output value among them.

5.6 Clustering

Clustering is a key technique in unsupervised machine learning, aiming to group similar data points together. It's all about ensuring that items within a cluster are more alike than those in different clusters. This approach is great for uncovering hidden patterns in data. When it comes to checking the quality of an embedding, clustering can be a handy tool. By forming clusters from embedded data, we can see how effectively similar structures come together. A top-notch embedding should make sure that data points from the same structure cluster closely. So, by looking at how well clustering works, we can gauge the strength of the embedding.

5.6.1 K-Means Clustering

K-Means [24] is a popular clustering method recognized for its straightforwardness and speed. It works by dividing a dataset into 'K' unique, separate groups (or clusters) based on how close each data point is to the cluster's center, termed the centroid. The method repeatedly places each data point with the closest centroid and then updates the centroid's position based on the points in its cluster. This cycle repeats until the centroids no longer move significantly. Though K-Means is great

for clusters that are roughly spherical and similar in size, deciding on the best number of clusters (K) in advance can be tricky.

5.6.2 DBSCAN

DBSCAN [7] is a clustering method that identifies dense regions in data, considering sparse areas as outliers. Unlike K-Means, DBSCAN doesn't need a pre-defined number of clusters. It works on the principle that a cluster is a high-density area in data, surrounded by less dense regions. The algorithm is steered by two key parameters: the minimum points needed for a region to be dense and a distance measure determining how close points should be to create a cluster. DBSCAN shines in handling datasets where clusters have varying shapes but similar densities.

5.6.3 Spectral Clustering

Spectral clustering[23] is a method that emphasizes reducing the dimensionality of data using the eigenvalues of a similarity matrix. By constructing a similarity graph, where nodes represent data points and edges carry weights based on point similarities, the technique transforms the original space. Utilizing the eigenvectors of the graph's Laplacian, it creates a more compact and manageable representation. In this reduced space, traditional clustering methods like K-Means become more effective. Spectral clustering's strength lies in its ability to handle complex cluster structures, especially non-convex clusters, making it a valuable tool for diverse applications.

6 Methods

This research dives into the complexities of embedding byte sequences, focusing particularly on the extraction of structures containing SSH keys for machine learning purposes. The varied uses of OpenSSH introduce distinct challenges due to potential variations in the created embeddings. Given the wide array of SSH key dimensions and OpenSSH's intricate operations, maintaining the embeddings' stability and consistency is vital. In this methodological section, we will detail various embedding methods, present a framework for their assessment through a classifier model, and suggest another strategy to verify the embeddings' coherence between the different OpenSSH usage and key sizes.

6.1 Embedding coherence

After completing the classification task, our focus shifts to evaluating the coherence of the embedding across different applications of OpenSSH and various key sizes. To accomplish this, we will utilize a clustering model, specifically DBSCAN, which is well-suited for scenarios where the number of clusters is uncertain. Our objective is to determine if the formed clusters demonstrate coherence, signifying the proximity of memory structures containing SSH keys. This analysis also encompasses an assessment of the underlying embedding method's consistency across various uses of SSH and key sizes, illustrating its ability to capture significant patterns and relationships related the the SSH keys.

In the following section, we will delve deep into the methodologies and techniques utilized to construct these embeddings, offering a comprehensive insight into the fundamental building blocks of our study.*

6.2 Dataset

The dataset at the core of this thesis, as previously introduced (see 5.1.3), consists of heap dump raw files related to different OpenSSH use cases and versions. Each heap dump file is paired with a JSON annotation file created by the dataset's creators. These JSON files provide extra information about the heap dump, especially regarding encryption keys. In this section, we will explain our exploration of the dataset, aiming to better comprehend its content and nuances.

6.2.1 Origin

The dataset is derived from heap dumps that capture various OpenSSH usage scenarios. These scenarios encompass four distinct SSH interactions: a straightforward client connection to the server followed by an immediate exit, port-forwarding, secure copying, and SSH shared connection. The heap dumps span different OpenSSH versions and a range of key sizes, from 16 to 64 bytes. These dumps were generated using the SmartKex tool [9]. The data collection was conducted on a mini PC equipped with an AMD Ryzen 5500U processor, 16GB of RAM, and a 1TB NVMe SSD, running Debian 11 as its operating system.

6.2.2 Estimating the dataset balancing for key prediction

In this part, our primary objective was to assess the balance of the dataset for key prediction and identify the challenges associated with it.

To begin, we aimed to gain an understanding of the dataset's scale. We utilized a code snippet 6.1 to count all the files within the dataset, revealing a total of 208,745 files. However, it was imperative to recognize that JSON files, which served as annotation files, were not to be considered part of the raw bytes for embedding. Consequently, these JSON files were excluded from our count to provide a more accurate representation of the dataset's size.

```
1         find . -type f | wc -l
```

Code 6.1: Count all dataset files

Following this, we employed another code snippet 6.2 to specifically count the heap dump raw files, excluding JSON files. This count indicated a total of 103,595 heap dump raw files, which constituted the primary focus of our analysis.

```
1         find . -type f -name "*.raw" | wc -l
```

Code 6.2: Count heap dump raw dataset files

To gain further insights into the dataset, we determined its size while excluding annotation files 6.3. The calculated dataset size amounted to 18,067,001,344 bytes.

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

Code 6.3: Get the size of the dataset

Considering the nature of the dataset, which featured a maximum of six keys per file, each with a maximum size of 64 bytes, we conducted a rough estimate. We determined that the maximum number of bytes relevant for searching across the dataset was $6 * 64 * 103595 = 39780480$. This calculation accounted for approximately 0.22% of the dataset's total size.

Lastly, it is crucial to acknowledge that the dataset exhibited a significant imbalance and is very large. To address this challenge effectively, strategies were implemented to ensure robust, unbiased analyses, and scalability.

Annotations

The annotations files are essential to understand the data and how best to utilize them for the study. Each heap dump corresponds to one specific JSON file. To view the contents of these JSON files in a more organized manner, one can reference the method provided at 6.4. For a clearer understanding, an extract of the JSON annotation from the file located at `./Training/client/V_7_8_P1/16/13116-1644920217.json` is available at 6.5.

```
python3 -m json.tool file.json
```

Code 6.4: pretty print JSON

```
{
    /* file ./Training/client/V_7_8_P1/16/13116-1644920217.json
       */
    "SSH_STRUCT_ADDR": "5619dd7e5570",
    "SESSION_STATE_ADDR": "5619dd7e5df0",
    "KEY_A_ADDR": "5619dd807f40",
    "KEY_A_LEN": "12",
    "KEY_A_REAL_LEN": "12",
    "KEY_A": "34fbe182e76c49a617a93e2e",
    /*...*/
    "KEY_E_ADDR": "5619dd808000",
    "KEY_E_LEN": "0",
    "KEY_E_REAL_LEN": "0",
    "KEY_E": "",
    "KEY_F_ADDR": "5619dd807fd0",
    "KEY_F_LEN": "0",
    "KEY_F_REAL_LEN": "0",
    "KEY_F": "",
    "HEAP_START": "5619dd7e3000"
}
```

Code 6.5: An extract of the JSON annotations

Within these annotation files, several critical pieces of information are present. The "SSH_STRUCT_ADDR" and "SESSION_STATE_ADDR" denote the addresses of vital openSSH structures. These addresses are pivotal in gauging the embedding coherence across different openSSH uses and key sizes. If the embeddings of these structures display similarity across various key sizes and openSSH usages, it signifies the embedding's coherence.

Other significant annotations such as "KEY_A_ADDR", "KEY_A_LEN", "KEY_A_REAL_LEN", and "KEY_A" detail the address, length, and value of the key A. In general, six of these annotations can be found for each heap dump. Notably, the "HEAP_START" annotation, along with the length of the heap dump, is of paramount importance. This annotation signifies the starting address of the heap dump. This information not only aids in pinpointing addresses in the heap dump for structures and pointers, but also refines the heuristic used in detecting pointers 5.1.5. By leveraging the "HEAP_START" information, one can verify if a pointer is pointing within the heap dump boundaries. As a practical illustration, deducing the address of key A within the heap dump can be achieved by subtracting "HEAP_START" from "KEY_A_ADDR".

However, it's noteworthy that some of these annotation files may be corrupted. Therefore, it's imperative to verify the integrity of each file before its use. In instances where keys are corrupted, such as "KEY_E" and "KEY_F" having no recorded values in the extract found at 6.5, it's advised either to remove the corrupted keys or discard the entire file if the data cannot be salvaged. Armed with this

understanding, the next logical step would be to leverage this dataset to formulate embeddings and subsequently evaluate their performance.

6.2.3 Malloc header usage and structures detection

As discussed in 5.1.5, subsequent to an initial 8-byte block of zeros, we anticipate the allocation of the first data structure at the heap's commencement. As illustrated in 6.1, this data structure spans a size of 5102000000000000_{16LE} (in little-endian hexadecimal notation) or 593_{10} bytes. The presence of an odd number arises from the LSB being set to 1, signifying that the block is allocated (as a flag). Consequently, the actual size of the structure is $593_{10} - 1_{10} = 592_{10}$ bytes, which aligns with an 8-byte boundary.

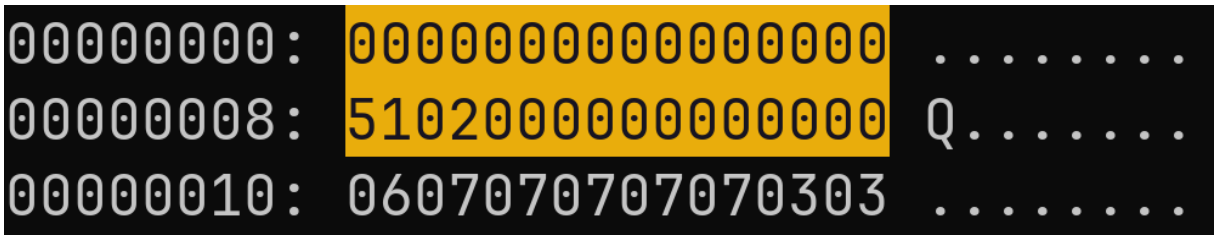

Figure 6.1: Attempt at malloc header detection in *Training/basic/V_7_8_P1/16/5070-1643978841-heap.raw*, at heap start.

Given the allocator's sequential chunk allocation approach, the subsequent chunk's anticipated allocation address is calculated as $5102000000000000_{16LE} + 592_{10} + 8_{10}$. The additional 8 accounts for the malloc header block, resulting in an address of $5882193a34560000_{16LE}$.

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 6.2:

Figure 6.2: Attempt at malloc header detection in *Training/basic/V_7_8_P1/16/5070-1643978841-heap.raw*, at index $592_{10} = 250_{16}$.

At this point, we observe a zero block, succeeded by a potential malloc header at address 258_{16}. By replicating this process, we can devise an algorithm to identify malloc headers, and consequently, the structures within the heap dump file.

Algorithm 6.6 Malloc Header Detection Algorithm

```
1:  procedure MallocHeaderDetection(heapDumpFile)
2:      position ← 0
3:      while position < FileSize(heapDumpFile) do
4:          block ← Read8Bytes(heapDumpFile, position)
5:          if block ≠ 0 then
6:              size ← ConvertToSize(block) − 1                ▷ −1 due to the flag
7:              Assert size  mod 8 = 0                         ▷ Check if the size is 8-bytes aligned
8:              position ← position + 8 + size                 ▷ Leap over data structure. + 8 for the header.
9:          else
10:             position ← position + 8
11:         end if
12:     end while
13: end procedure
```

The underlying concept of the malloc header detection algorithm is straightforward. Beginning at the start of the heap dump file, we search for the initial non-zero block. Subsequently, we infer that the following block represents a malloc header. This block is converted into a size, allowing us to skip over both the data structure and its header. This procedure continues iteratively until the file's conclusion.

6.3 Embedding

From the Zenodo dataset5.1.3, we've isolated distinct memory structures within the raw heap dump files. These structures possess diverse sizes, necessitating the use of an embedding method for classification. Fortunately, a distinguishing feature of each memory structure is the presence of a header, containing vital information such as the structure's size in bytes. To precisely pinpoint the boundaries of each memory structure, we sequentially parse through the raw heap dump files. Beginning the parsing process from the first non-null byte, identified as the header, serves as a marker for the initiation of a new structure. The size data within this header is then leveraged to calculate the exact length of the structure, allowing for the extraction of its entire raw byte data while determining the start of the subsequent one.

Our next objective centers on the conversion of raw byte data into fixed-size embeddings (5.2, 5.3), a pivotal step in preparing them for utilization in machine learning applications. Ensuring uniformity in embedding size across all memory structures holds paramount significance. Consistency in embedding dimensions is vital to empower machine learning algorithms for efficient data processing and analysis. This uniformity not only simplifies the integration of memory structures with varying sizes into a coherent classification framework but also acts as a defense against the adverse effects of the curse of dimensionality—a phenomenon that can introduce computational complexities and heighten the risk of overfitting in high-dimensional data spaces. Striking this equilibrium is essential, achieved by maintaining reasonably low embedding dimensions, fostering both efficient data processing and the preservation of essential information within the raw byte data. It's important to note that initially, each embedding will include the structure's file and the structure's address in the file. However, these details will be removed during the machine learning phase (quality or coherence) as the embedding aims to be free of key size or OpenSSH uses. Their presence will serve as a means to test coherence later in our analysis.

6.3.1 Statistical embedding

Understanding the fundamental concepts of statistical embeddings enables us to delve deeper into the sophisticated processes and practical applications that underscore their significance in embedding tasks. By utilizing statistical techniques, data from high-dimensional spaces is condensed, preserving the inherent probabilistic connections and essential patterns as much as possible.

6.3.1.1 N-gram values

In reference to section 5.2.2, we adopt the use of n-gram values, specifically focusing on the frequency of byte combinations. However, an implication of this approach is that it leads to an exponentially high dimensional space. For instance, with a 2-gram, the potential values amount to $256 * 256 = 65536$. Given the extensive dimensionality, we have opted for combinations of bits rather than bytes. This change substantially reduces the space required; a 2-gram, in this case, would only amount to $2 * 2 = 4$ values.

Switching to bit combinations aligns well with our objectives. Our main interest is in the frequency patterns of n-gram values rather than the specific n-gram values themselves. This is because our core aim is to identify SSH keys, which inherently display frequencies for all combinations due to their random nature.

In our approach, we utilize 1-gram, 2-gram, and 3-gram values. As a result, our dimensional space is confined to 14 dimensions, as calculated by $2*2*2+2*2+2 = 14$. We believe this is an optimal trade-off, striking a balance between the size of the space and the richness of the information it encapsulates.

6.3.1.2 Other statisticals values

In our approach, several metrics are employed to analyze the data. Specifically, we utilize the mean as detailed in 5.2, the standard deviation as found in 5.4, the MAD from 5.3, the skewness as outlined in 5.5, the kurtosis referenced in 5.6, and the Shannon entropy from 5.1. These metrics, when collectively considered, provide a comprehensive understanding and embed a plethora of information about the data at hand.

It's imperative to note a particular aspect of our analysis concerning the standard deviation. There are instances where the standard deviation registers a value of zero. Such an occurrence is indicative of data consistency. Concurrently, in such scenarios, both the kurtosis and skewness are undefined. When faced with this situation, our course of action is to dismiss the structure from our analysis. The rationale behind this is straightforward: a consistent structure would likely not be pertinent to our exploration, especially when our aim is to identify patterns characteristic of an SSH key, wich are random by nature.

6.3.1.3 Statistical embedding

We employ then a combination of n-gram values and other statistical metrics to construct the vectors for each structure. The n-gram approach contributes 14 distinct values to the vector. Simultaneously, the supplementary statistical metrics, which encapsulate measures of the mean, standard deviation, MAD, skewness, kurtosis, and Shannon entropy, introduce an additional 6 values. Consequently, the resultant vector for each structure comprises a total of 20 values.

6.3.2 Graph embedding

In this section, we shift our focus towards the creation and embedding of graphs derived from the heap dump data. The process of graph creation involves structuring the data in a way that captures the relationships and connections between the structures and their pointers. Subsequently, we will transform this graphs into low-dimensional vector representations, enabling the application of machine learning techniques to identifying structures containing ssh keys.

6.3.2.1 Graphs creation

Our graph construction is a meticulously organized process aimed at representing the intricate relationships present within the heap dump data. Comprising three distinct node types - structures, pointers, and value nodes - this graph provides a comprehensive view of the data's structure. Our approach commences with the sequential parsing of the heap dump data, enabling the identification of essential structures central to our analytical objectives. These structures form the core nodes of our graph. To establish connections between these structures and their contained data, we further break down each structure into 8-byte blocks. These blocks are then translated into value nodes within the graph, serving as connectors bridging the data structures to their specific data. An heuristic approach, grounded in REGEX, is employed to identify valid pointers within the heap dump data, with pointers representing a subset of value nodes, indicating legitimate pointers references. The scrupulously established connections between structures, value nodes, and pointers ensure that the graph accurately mirrors the intricate relationships found within the heap dump data. This comprehensive graph construction process is efficiently implemented in Rust, making effective use of the Petgraph library to handle the complexities of heap dump data and graph representation, offering superior efficiency compared to a Python-based implementation.

In the following image 6.3, we can see the structures nodes representing in blue, containing pointers nodes in orange and value nodes nodes in gray.

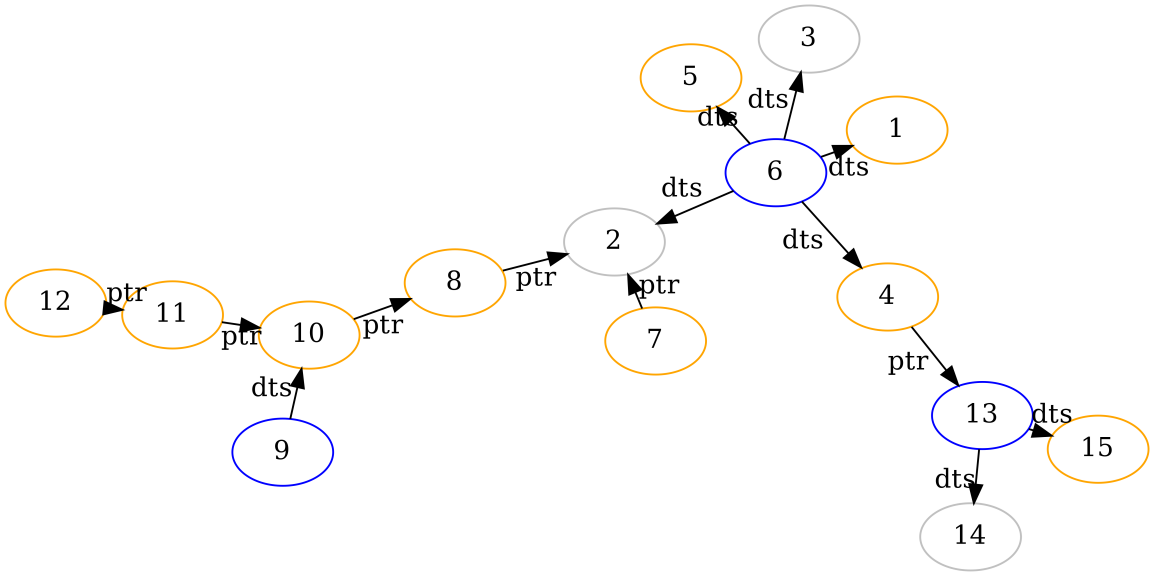

Figure 6.3: Graph creation process

After the construction of the graph, we can use graphviz (and the DOT language)[6] to visualize the graph, using the command :

```
sfdp -Gsize=67! -Goverlap=prism -Tpng dot_file > image.png
```

The following image is an example of the creation of the graph from the file `./Training/Training/scp/V_7_8_P1/16/302-1644391327-heap.raw` without value nodes to enhance clarity.

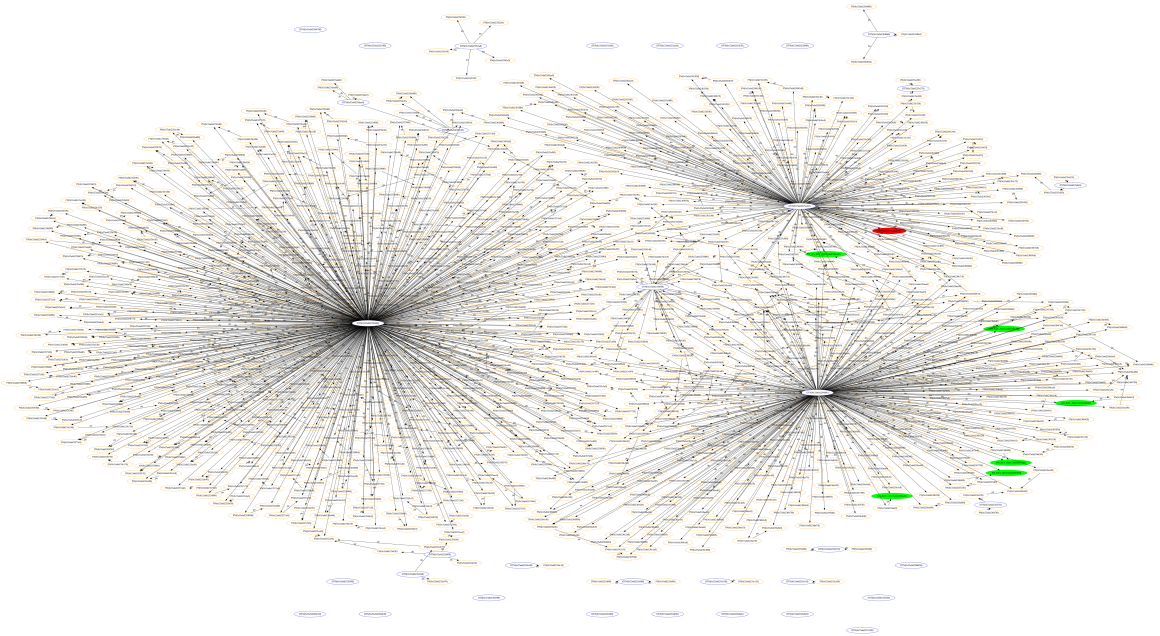

Figure 6.4: Graph example

6.3.2.2 Graphs embedding

Our next step is to uncover deeper insights and semantic understanding from our constructed graph, focusing on semantic embedding. This is the process through which we reshape our graph into a low-dimensional vector space, with each vector acting as a repository for a structure's immediate neighborhood. Through this transformative journey, our aim is to forge vector representations that empower the application of cutting-edge machine learning techniques.

To create a concise yet informative representation, considering both structure-to-member and pointer-based connections, we meticulously count the number of pointers and structures directly referencing a specific structure's members. This initial count provides valuable insights into the structure's immediate context. However, we doesn't stop there; we expand this representation by including counts of pointers and structures pointing to those preceding nodes, allowing us to capture deeper layers of context. This recursive process continues until we reach a predetermined depth. Furthermore, we initiate a parallel analysis in reverse, meticulously tracing connections by following pointers from the initial structure to capture its children, recursively delving deeper until we reach the specified depth. We can see the algorithm here 6.7. The result is a low-dimensional vector that intricately encodes the structure's neighborhood, offering a comprehensive view of its relationships and contextual significance within the graph.

Algorithm 6.7 Generate Ancestor/Children Embedding

```
function GenerateNeighborsDTN(structure_node, dir)
    ancestor_nodes ← an empty set
    children ← graph.neighbors_directed(structure_node, OUT)   ▷ Get members of the structure
    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_dtn ← 0
        nb_ptr ← 0
        for current_node in current_nodes do
            if node is DataStructureNode then          ▷ Update number of structures and pointers
                nb_dtn ← 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_dtn)                          ▷ Add number of data structures
        result.append(nb_ptr)                          ▷ Add number of pointers
    end for
    return result
end function
```

We can apply this algorithm to every structure within each graph, delving to a depth of 8, which produces an embedding of 32 units: 8 for ancestor pointers, 8 for ancestor structures, 8 for child pointers, and 8 for child structures. To accurately represent the structure's neighborhood, it's crucial not to omit details about its members. Thus, we incorporate the count of pointers in the members and the structure's dimensions. This results in a final embedding size of 34 - 32 for the neighborhood and an additional 2 for the structure size and pointer count. However, there are inherent challenges with this embedding. It tends to get polluted by the value node, which often lacks significant meaning. Moreover, the relationships between the structures are intricate, and there's potential to represent them in a more straightforward manner, as show in the next section.

6.3.2.3 Updated graph

Recognizing these challenges and the need for a clearer representation, we embarked on a series of refinements. Our approach focuses on enhancing the last graph by preserving the structure nodes and their interconnections via pointers. To simplify the visualization, we've decided to eliminate both the value nodes and the pointer nodes. In addition, the relationships that previously connected the pointer nodes to the value nodes will now link directly to the structure nodes, with the added detail of weighted edges. This strategy is driven by our aspiration to offer a more lucid graph, significantly reducing any

extraneous noise, as show in the figure 6.5, the representation of the file *14911-1644326802*.

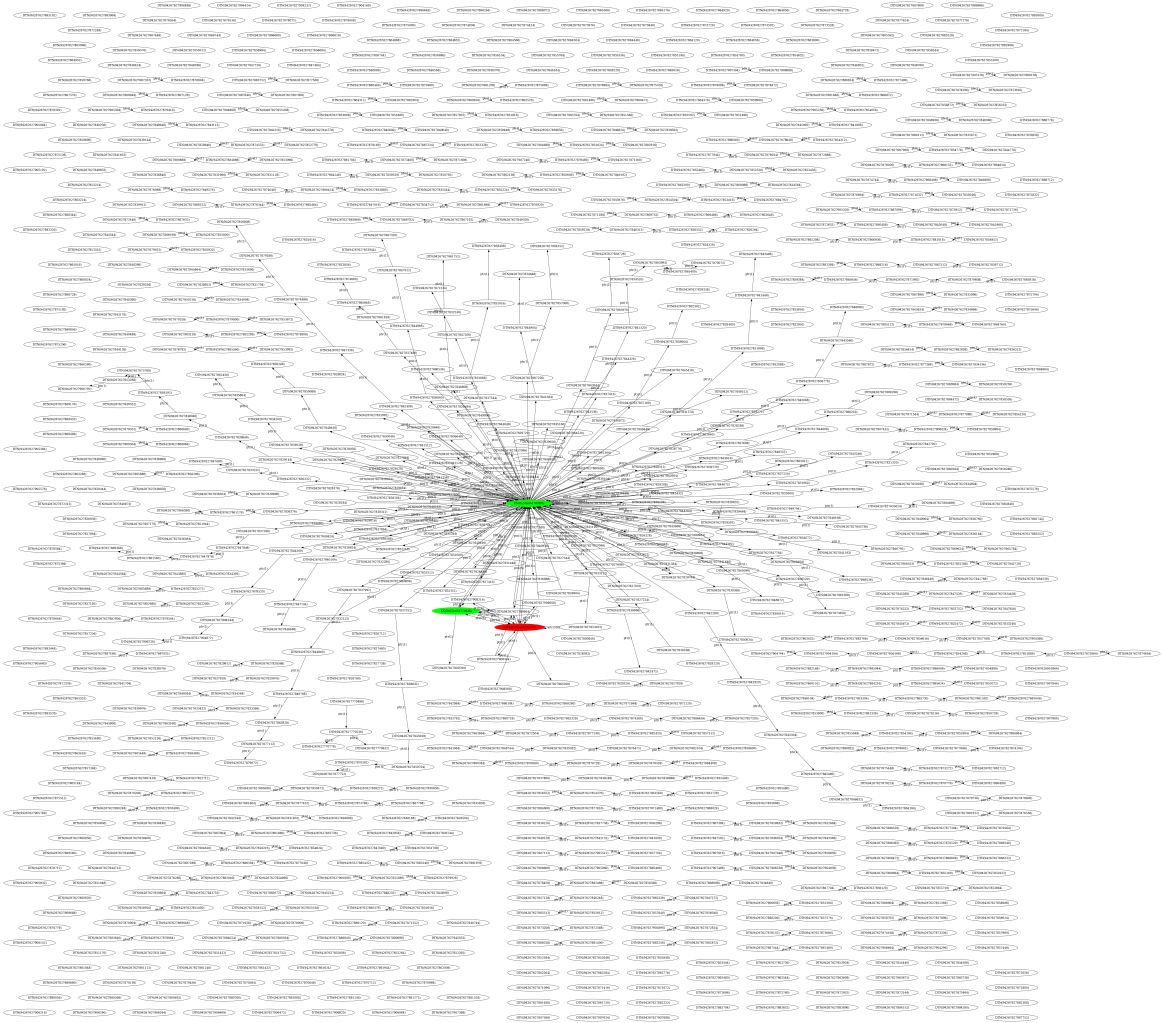

Figure 6.5: Updated graph

6.4 Embedding quality

The quality of embeddings is paramount in machine learning, particularly when the objective is to identify specific structures within data, such as the ones holding SSH keys. It becomes essential to juxtapose the performances of all embeddings in this context. An optimal embedding should proficiently discern the structures containing SSH keys across the entire spectrum of openSSH use cases and for every conceivable key size. This necessitates the utilization of the complete dataset, with the training subset dedicated to model training and the validation subset for testing. Addressing this from a machine learning classification perspective, the random forest model, as elucidated in 5.5.3, emerges as the classifier of choice.

To ensure fairness and comparability among the embeddings, we employ the Pearson correlation method 5.5.1.1 to limit the selection to the top 8 correlations, thereby narrowing down our analysis to the most influential features. The dataset is notably imbalanced 5.5.2, primarily stemming from the rarity of memory structures containing SSH keys, our specific target of interest, within the overall dataset. This rarity results in a significant class imbalance, where the majority of memory structures do not contain SSH keys. To counteract potential bias toward the majority class, we will implement random undersampling as a resampling strategy, particularly given our very large dataset. This approach will enable our model to accurately classify both majority and minority classes without being overwhelmed by the sheer volume of data. We will then employ a Random Forest model 5.5, renowned for its robustness and suitability for high-dimensional data, to carry out the classification task. Our evaluation will rely on metrics such as precision, recall, F1 score, and others to identify the most effective representation for precise classification.

6.4.1 Feature Selection and Dataset Challenges

In the quest for fairness across various embeddings and to circumvent the curse of dimensionality, it's imperative to maintain a uniform feature count across all embeddings. This is where feature engineering shines. The Pearson correlation method, elaborated in 5.5.1.1, is harnessed to meticulously select the 8 most salient features for each embedding. This count is a judicious compromise, ensuring the features are both succinct in number and information-rich. However, the dataset presents its own set of challenges. The instances of structures containing SSH keys are dwarfed by those devoid of them, leading to a pronounced dataset imbalance. To counteract this skewness, the random undersampling technique, as referenced in 5.5.2, is employed.

6.4.2 Implementation and Evaluation Metrics

The implementation leans heavily on the scikit-learn library [27] in Python, which provides the tools for the random forest classifier, Pearson correlation, and the random undersampling algorithm. Concurrently, the pandas library is indispensable for the efficient loading and manipulation of the dataset. Before diving into the analysis, it's crucial to ensure the embedding's integrity. This involves a rigorous sanity check, especially given the potential for corruption, such as NaN values. To guarantee

the reproducibility of results, a consistent random seed is employed for both the random forest classifier and the random undersampling algorithm. For a comprehensive evaluation, the Pearson correlation matrix is preserved for each embedding. Moreover, a suite of metrics, including precision, recall, f1-score, AUC, and the confusion matrix (encompassing true positives, true negatives, false positives, and false negatives), is meticulously saved for every embedding.

7 Results

Describe the experimental setup, the used datasets/parameters and the experimental results achieved

8 Discussion

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

9 Conclusion

Summarize the thesis and provide a outlook on future work.

10 Ressources

TODO : make transition

10.1 hardware

My primary workstation is an *Aspire 5* laptop, equipped with:

- **CPU:** 11th Gen Intel i5-1135G7 (8) @ 4.200GHz
- **GPU:** Intel TigerLake-LP GT2 [Iris Xe Graphics]
- **Memory:** 16GB

However, this laptop, despite its decent specifications, proved inadequate for processing the entire dataset. Simple machine learning experiments using a Python script would have stretched over a week. Even when we transitioned to more optimized Rust programs, the processing time exceeded 10 hours. While I managed to run minor tasks and scripts on this laptop, the bulk of the experiments necessitated a more powerful server.

Recognizing this need, I was granted access to a high-performance development server in the later stages of the thesis, around August 2023. The server, an *AS-4124GS-TNR*, boasts the following specifications:

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

Operating on *Ubuntu 20.04.6 LTS*, this server became the primary platform for the machine learning experiments, given its superior computational capabilities compared to the *Aspire 5* laptop. This invaluable resource was generously provided by the Department of Computer Science at *Universität Passau*, particularly under the guidance of the Chair of Data Science led by Prof. Dr. Michael Granitzer. I extend my sincere appreciation for their unwavering support.

A Code

B Math

C Dataset

Acronyms

BFD Byte Frequency Distribution. 3, 8, 9

CNN Convolutional Neural Networks. 3, 10, 12, 13

GRU Gated Recurrent Units. 3, 11, 12

KNN K-Nearest Neighbors. 19

LSB Least Significant Bit. 25

LSTM Long Short-Term Memory. 3, 11, 12

PCA Principal Component Analysis. 18

RCNN Recurrent Convolutional Neural Network. 11

REGEX regular expressions. 6, 29

RNN Recurrent Neural Networks. 3, 10–12

SCP secure copy. 6

Seq2Seq Sequence-to-Sequence. 12, 13

SMOTE Synthetic Minority Over-sampling Technique. 18

SSH Secure Shell. 1, 3, 4

SVM Support Vector Machines. 19

t-SNE t-Distributed Stochastic Neighbor Embedding. 18

VMI Virtual Machine Introspection. 1

Glossary

pointer In our study, pointers are characterized as sequences of hexadecimal numbers that reference distinct memory addresses. These sequences can be recognized using the following regular expression: `"[0-9a-f]{12}0{4}"`. 6, 24, 28–31

structure In our study, structures are defined as a series of bytes that are allocated in the heap. These structures are allocated using the `calloc` function and begin everytime by a *malloc header*. 7, 24, 28–31, 33

value node In our study, value nodes represent 8-byte blocks of data that are contained within a structure.. 29, 30

References

[1] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. *An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling*. Apr. 19, 2018. arXiv: 1803.01271[cs]. URL: http://arxiv.org/abs/1803.01271 (visited on 08/23/2023).

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

[3] Meghan K. Cain, Zhiyong Zhang, and Ke-Hai Yuan. „Univariate and multivariate skewness and kurtosis for measuring nonnormality: Prevalence, influence and estimation". In: *Behavior Research Methods* 49.5 (Oct. 2017), pp. 1716–1735. ISSN: 1554-3528. DOI: 10.3758/s13428-016-0814-1. URL: http://link.springer.com/10.3758/s13428-016-0814-1 (visited on 08/30/2023).

[4] 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).

[5] 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.

[6] John Ellson et al. „Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools". In: *Graph Drawing Software*. Ed. by Michael Jünger and Petra Mutzel. Red. by Gerald Farin et al. Series Title: Mathematics and Visualization. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 127–148. ISBN: 978-3-642-62214-4 978-3-642-18638-7. DOI: 10.1007/978-3-642-18638-7_6. URL: http://link.springer.com/10.1007/978-3-642-18638-7_6 (visited on 09/11/2023).

[7] Martin Ester et al. „A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *KDD-96* (Aug. 2, 1996).

[8] Christofer Fellicious et al. *Machine Learning Assisted SSH Keys Extraction From The Heap Dump*. Version 0.1. Aug. 15, 2022. DOI: 10.5281/ZENODO.6537904. URL: https://zenodo.org/record/6537904 (visited on 09/06/2023).

[9] Christofer Fellicious et al. *SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump*. Sept. 13, 2022. arXiv: 2209.05243[cs]. URL: http://arxiv.org/abs/2209.05243 (visited on 08/17/2023).

[10] Jonas Gehring et al. „Convolutional Sequence to Sequence Learning". In: *Facebook AI Research* (July 25, 2017). URL: https://arxiv.org/pdf/1705.03122.pdf.

[11] Luke Hiester. „File Fragment Classification Using Neural Networks with Lossless Representations". In: *East Tennessee State University* (May 2018). (Visited on 08/21/2023).

[12] G. E. Hinton and R. R. Salakhutdinov. „Reducing the Dimensionality of Data with Neural Networks". In: *Science* 313.5786 (July 28, 2006), pp. 504–507. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.1127647. URL: https://www.science.org/doi/10.1126/science.1127647 (visited on 08/30/2023).

[13] 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).

[14] Aidan Hogan et al. „Knowledge Graphs (Extended)“. In: *ACM Computing Surveys* 54.4 (May 31, 2022), pp. 1–37. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3447772. arXiv: 2003.02320[cs]. URL: http://arxiv.org/abs/2003.02320 (visited on 09/08/2023).

[15] 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*. 2014 Science and Information Conference. Aug. 2014, pp. 372–378. DOI: 10.1109/SAI.2014.6918213.

[16] Udayan Khurana, Horst Samulowitz, and Deepak Turaga. *Feature Engineering for Predictive Modeling using Reinforcement Learning*. Sept. 21, 2017. arXiv: 1709.07150[cs,stat]. URL: http://arxiv.org/abs/1709.07150 (visited on 08/30/2023).

[17] Mario Koppen. „The curse of dimensionality“. In: 1 (2000), pp. 4–8.

[18] 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).

[19] 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).

[20] Siwei Lai et al. „Recurrent Convolutional Neural Networks for Text Classification“. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 29.1 (Feb. 19, 2015). ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v29i1.9513. URL: https://ojs.aaai.org/index.php/AAAI/article/view/9513 (visited on 08/23/2023).

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

[22] 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. Issue: 7. 2021, pp. 6418–6425. URL: file:///home/onyr/Downloads/16796-Article%20Text-20290-1-2-20210518.pdf.

[23] Ulrike von Luxburg. *A Tutorial on Spectral Clustering*. Nov. 1, 2007. arXiv: 0711.0189[cs]. URL: http://arxiv.org/abs/0711.0189 (visited on 09/05/2023).

[24] J Macqueen. „SOME METHODS FOR CLASSIFICATION AND ANALYSIS OF MULTIVARIATE OBSERVATIONS“. In: *MULTIVARIATE OBSERVATIONS* VOL. 5.1 (1967).

[25] 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.

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

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

[28] 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).

[29] Dr D Ramyachitra and P Manikandan. „IMBALANCED DATASET CLASSIFICATION AND SOLUTIONS: A REVIEW“. In: *International Journal of Computing and Business Research* 5.4 (2014).

[30] 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.

[31] Stewart Sentanoe and Hans P. Reiser. „SSHkex: Leveraging virtual machine introspection for extracting SSH keys and decrypting SSH network traffic“. In: *Forensic Science International: Digital Investigation* 40 (Apr. 2022), p. 301337. ISSN: 26662817. DOI: 10.1016/j.fsidi.2022.301337. URL: https://linkinghub.elsevier.com/retrieve/pii/S2666281722000063 (visited on 08/17/2023).

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

[33] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks*. Dec. 14, 2014. arXiv: 1409.3215[cs]. URL: http://arxiv.org/abs/1409.3215 (visited on 08/23/2023).

[34] Ashish Vaswani et al. „Attention Is All You Need“. In: *Advances in Neural Information Processing Systems* 30 (2017), pp. 5998–6008. (Visited on 08/23/2023).

[35] Michel Verleysen and Damien François. „The Curse of Dimensionality in Data Mining and Time Series Prediction“. In: *Computational Intelligence and Bioinspired Systems*. Ed. by Joan Cabestany, Alberto Prieto, and Francisco Sandoval. Red. by David Hutchison et al. Vol. 3512. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 758–770. ISBN: 978-3-540-26208-4 978-3-540-32106-4. DOI: 10.1007/11494669_93. URL: http://link.springer.com/10.1007/11494669_93 (visited on 08/30/2023).

[36] 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: The 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. San Francisco California USA: ACM, Aug. 13, 2016, pp. 1225–1234. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939753. URL: https://dl.acm.org/doi/10.1145/2939672.2939753 (visited on 09/11/2023).

[37] Donald J Wheeler. „Problems with Skewness and Kurtosis“. In: *Quality Digest Daily* (Aug. 1, 2011).

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

[39] 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.

Additional bibliography

[40] Walter T. Ambrosius, ed. *Topics in biostatistics*. Methods in molecular biology 404. OCLC: ocn159977868. Totowa, N.J: Humana Press, 2007. 528 pp. ISBN: 978-1-58829-531-6.

[41] *CERT/CC Vulnerability Note VU#13877*. URL: https://www.kb.cert.org (visited on 08/30/2023).

[42] Vivek Gite. *How To Reuse SSH Connection To Speed Up Remote Login Process Using Multiplexing*. nixCraft. Aug. 20, 2008. URL: https://www.cyberciti.biz/faq/linux-unix-reuse-openssh-connection/ (visited on 10/21/2022).

[43] 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*. Ed. by Jose Manuel Gomez-Perez, Ronald Denaux, and Andres Garcia-Silva. 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 (visited on 09/08/2023).

[44] 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.

[45] Weijie Huang and Jun Wang. *Character-level Convolutional Network for Text Classification Applied to Chinese Corpus*. Nov. 15, 2016. arXiv: 1611.04358[cs]. URL: http://arxiv.org/abs/1611.04358 (visited on 08/17/2023).

[46] Michael I Jordan and Tom M Mitchell. „Machine learning: Trends, perspectives, and prospects“. In: *Science* 349.6245 (2015). Publisher: American Association for the Advancement of Science, pp. 255–260. URL: https://www.science.org/doi/full/10.1126/science.aaa8415.

[47] 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. 1, 2021), pp. 387–396. ISSN: 0167-739X. DOI: 10.1016/j.future.2020.09.004. URL: https://www.sciencedirect.com/science/article/pii/S0167739X20303265 (visited on 08/30/2023).

[48] 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.* 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.

[49] 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).

Eidesstattliche Erklärung

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

Passau, October 4, 2023

_____

Lahoche, Clément Claude Martial