Lehrstuhl für Data Science

Utilizing Graph and Reinforcement Learning Algorithms for SSH Key Retrieval from Heap Memory Dumps

Masterarbeit von

Cyril GOMES

1. Prüfer	2. Prüfer
Prof. Dr. Michael Granitzer	Prof. Dr. Harald Kosch

April 18, 2024

Contents

Abstract

Secure Shell (SSH) is a cryptographic network protocol essential for secure machine-to-machine communication, particularly prevalent in today's internet-driven world. SSH ensures security by encrypting messages using keys that only the communicating machines can access, thus safeguarding against unauthorized decryption. In this work, we propose a novel approach for extracting these keys from a memory dump using Deep Reinforcement Learning (DRL). The appeal of this method lies in its ability to learn to extract the keys using only a limited amount of data and to generalize effectively to new, unseen data, significantly reducing the challenge of generating large volumes of high-quality data for training. Therefore, when a new OpenSSH version is released, the model should be able to extract the keys from it without requiring further training or with minimal additional training.

Acknowledgments

First, I would like to express my deepest gratitude to my thesis advisors, Prof. Dr. Michael Granitzer and Christofer Fellicious, for their guidance, support, and encouragement throughout this research. Their expertise and insights have not only shaped this work but have also inspired me to push the boundaries of my knowledge and capabilities. I am equally thankful to Prof. Dr. Elod Egyed-Zsigmond, for his critical insights and constructive critiques, which have been crucial in refining this thesis. Additionally, I am grateful to my classmates and friends for their friendship and moral support, which have been invaluable during this journey. Lastly, I want to thank my family for all the love and patience they have shown me, which has truly kept me going through all the ups and downs.

List of Figures

List of Tables

1 Introduction

In an ultra-connected world where communication between people and devices has become seamless, the importance of securing these communications cannot be overemphasized, particularly for companies. Consequently, SSH, a protocol known for its strong security features, has become the preferred method for secure communication with remote devices. It is widely used in companies, and even by individuals, to securely access remote machines, transfer files, and execute commands. This protocol utilizes "**session keys**," which are temporary encryption keys used only for the duration of a session, to secure communications. These keys help ensure that even if data transmission is intercepted, it would remain protected.

However, it is important to note that SSH's popularity and capabilities also make it a target for exploitation by malicious actors. Such malicious actors could infect a device and use SSH to securely communicate with it, enabling them to perform various harmful actions. These could include stealing data or installing backdoors, all while remaining undetected by the owner of the device. For instance [AWK], a recent attack involved a backdoor in the open-source XZ/libzma library, which compresses and decompresses data [NIST2024; HSA; OSB]. In one of the releases of this library, malicious code was added. To summarize, this library, which is included in sshd (the OpenSSH server process), had malicious code that replaced one of the OpenSSH functions used to validate SSH keys. This replaced code checks for a particular signature (the one of the malicious actor) and if it is present, it bypasses the normal login process. Furthermore, in an effort to make it as undetectable as possible, the code included some "safety" mechanisms. Among other things, It would also detect the use of debugging tools, and if any are found, it would deactivate the backdoor.

Given the complexity and stealthiness of such attacks, the critical role of digital forensics becomes apparent. Digital forensics involves methods designed to efficiently gather and

analyze data from digital devices. The goal is to better assess the impact of a potential intrusion on the information system. Digital forensics investigations primarily aim to extract data or evidence from systems. However, it is equally important to conduct these investigations in a non-intrusive manner. This means that the investigation should neither alter the system in any way nor alert the potential attacker. Therefore, investigations that involve decrypting SSH communications can be particularly useful in the context of responding to malicious attacks or monitoring honeypot activities. It would allow investigators to understand the nature of a potential attack, what actions have been performed on the machine, and what data has been compromised, all without the attackers knowing that we are aware of their presence.

In this work, we aim to **extract SSH session keys** that are essential for decrypting encrypted communications. From the need for minimal intrusion, we decided to extract these keys from the heap memory dump of an OpenSSH process. This approach ensures the preservation of the system's integrity, as it does not require any modification to the system or the process source code. It also has the added benefit of being able to offload the decryption process to a more powerful machine, and therefore reduce the impact on the system being investigated. To navigate and extract the session keys from this complex and rather large memory structure, we employ a novel approach that converts this heap into a graph representation and makes use of Deep Reinforcement Learning to extract the SSH session keys from this graph. This approach has the advantage of requiring relatively small amounts of training data and has the potential to generalize across different versions of OpenSSH. Moreover, adapting this approach to handle newer versions of OpenSSH is relatively straightforward, as it only requires to be "fine-tuned" on the new version.

Reinforcement Learning and Deep Reinforcement Learning in particular, have been used in a wide variety of fields, from playing games [Sil+16], to robotics [Haa+24], to finance [ZZR19], and it has also been used in the field of cyber security [SSR23; BGR19].

In this work, we contribute to the field of digital forensics by proposing:

- A model that aims at predicting from the heap graph, the starting point of where the Reinforcement Learning Agent should start exploring.
- A model that aims at predicting from the heap graph, which keys are being used by the SSH process.

- A novel approach that combines the use of Graph Neural Networks and Deep Reinforcement Learning to extract SSH Session Keys.

Using this approach, we demonstrate that our method can efficiently explore the graph and find the SSH Session Keys, even when trained on limited data.

1.1 Research Questions

The main research questions we are trying to answer are the following :

- Is it possible to extract SSH keys when very little training data is present?
- Is there a suitable data structure that can represent the semantic relationships within the structures in a heap?

1.2 Structure of the Thesis

In this thesis, we begin by establishing the foundational concepts and contextual background necessary for this research, as detailed in chapter 2. Next, in chapter 3, we review related works to deepen our understanding of the problem, explore previous approaches, and illustrate the potential benefits of our method. chapter 4 outlines our methodology, delving into the specifics of data collection, the transformation of heap memory dumps into graph representations, and the application of these graphs within our Deep Reinforcement Learning algorithm. In chapter 5, we provide a thorough analysis of the challenges we faced during implementation and the solutions we employed. Our experimental results are discussed in chapter 6, where we evaluate the effectiveness of our approach. Finally, we conclude this thesis in chapter 7 by synthesizing our responses to the primary research questions and reflecting on the implications of our findings.

2 Background

This section provides background relevant to the thesis. We explain how SSH works by highlighting some key elements. We also give a short introduction to Reinforcement Learning, And Graph Neural Networks.

2.1 Secure Shell (SSH)

SSH (Secure Shell) [YL06], a network protocol established by Tatu Ylönen in 1995, revolutionized secure online communication. Its development primarily aimed to address the security limitations of non-encrypted protocols, such as Telnet, which was prevalent at the time.

While SSH is versatile, offering services such as secure file transfer, its primary application is in facilitating secure logins and shell access on remote machines and ensuring safe and confidential system administration. SSH uses asymmetric cryptography for user authentication. In a standard SSH session, key management involves four keys: a pair of Public and Private Keys for each communicating party, essential for the encryption and decryption processes. During communication, messages are encrypted using the recipient's Public Key, which can only be decrypted by the corresponding Private Key held exclusively by the recipient. This method ensures that intercepted communications remain unreadable to unauthorized parties. This cryptographic approach effectively guarantees the confidentiality and integrity of the data. It safeguards against external interception, as unauthorized entities, lacking the corresponding Private Key, cannot decrypt the secure communication. Among various implementations of SSH, OpenSSH stands out as a widely adopted open-source version, renowned for its robust security features and global usage. A tool like OpenSSH abstracts the inner workings of the protocol and provides a smoother experience to the users.

The SSH protocol is built on top of TCP and follows the client-server model [RFC4253; RFC4252]. At first, the SSH client establishes a connection to the server, which then sends its public key to the client. If the client does not recognize the received public key, it sends an alert. Now the client has to authenticate itself to the server. 2 methods are possible, either using a Password Authentication or a Key-Based Authentication. Password Authentication is straightforward, as its name implies, the client sends a password, which will be encrypted using the server's public key. The servers check if this password matches the stored one (usually hashed). A Key-Based Authentication involves a "trial", where the server sends this "trial" created using the client's public key. If the client manages to prove that it successfully decrypted the messages, he must then be the owner of the corresponding private key. Now that the client is authenticated to the server. Both the client and the server can use a method like the Diffie-Hellman algorithm to create a shared key, which will be used with standard hashing functions to encrypt the subsequent communication and assert its integrity. The client now can safely send data such as commands to the remote machine.

2.1.1 SSH Key Exchange and Sessions

Upon the establishment of the shared key K within an SSH session, it becomes feasible to generate a hash value h [RFC4253]. This hash value is derived by applying a cryptographic hashing function to a concatenation of various session-specific pieces of information. These typically include the public keys of both the server and client, the shared secret key K, and potentially additional data pertinent to the session. Furthermore, the concept of a *session_id* is introduced, which is initially assigned the value of h from the first key exchange of the session. It is crucial to note that while K and h may undergo recalculation during the ongoing session, the *session_id* remains constant. These three elements K, h, and the *session_id* serve in the derivation of a new set of cryptographic keys for each session, encompassing six distinct keys :

- Key A : $IV_{client2server}$, Initialization vector from client to server
- Key B : $IV_{server2client}$, Initialization vector from server to client
- Key C : $EK_{client2server}$, Encryption Key from client to server
- Key D : $EK_{server2client}$, Encryption Key from server to client
- Key E : $IK_{client2server}$, Integrity Key from client to server
- Key F : $IK_{server2client}$, Integrity Key from server to client

Thus, to accurately decrypt the transmissions within an SSH session, it is imperative to possess knowledge of either the set of values comprising the shared secret key K, the hash value h, and the $session\_id$, or, in the absence of access to these values, the appropriate combination of keys depending on the direction of the communication flow. Specifically, for client-to-server communication, Key A (Initialization Vector, client to server) and Key C (Encryption Key, client to server) are requisite. Conversely, for server-to-client communication, Key B (Initialization Vector, server to client) and Key D (Encryption Key, server to client) are essential. It is noteworthy that Key E (Integrity Key, client to server) and Key F (Integrity Key, server to client) do not play a direct role in the decryption process. In this work, we aim to extract keys A, B, C, and D, (Initialization Vector and Encryption Keys) as these keys are crucial for decrypting the SSH session data. We will refer to these keys as the SSH keys throughout this work.

2.1.2 OpenSSH

As previously stated, OpenSSH [OpenSSH; OSSHPort] is widely recognized as the predominant SSH implementation globally. This software suite encompasses two primary versions: the standard OpenSSH [OpenSSH], integrated natively into several major operating systems including its origin, the OpenBSD system; and Portable OpenSSH [OSSHPort], a versatile multi-platform variant. Portable OpenSSH, developed by the OpenBSD team in 1999, ensures compatibility across a diverse range of systems, notably including GNU/Linux. It is available for installation as a separate package on these systems. The split between the two versions comes down to meeting different needs: while the standard OpenSSH is optimized for OpenBSD's security-focused architecture, Portable OpenSSH extends this secure networking utility to a broader array of operating environments, maintaining the core principles of security and reliability. OpenSSH uses different encryption technologies, each encryption technology can have different key lengths, which can vary between 12 and 64 bytes.

2.2 Heap Memory

Heap Memory [Heap], often referred to as Dynamic Memory, operates in conjunction with Stack Memory. Stack Memory primarily stores data such as local variables, function

parameters, and return values. These data types are allocated to Stack Memory because they have well-defined lifetimes and can be deallocated at the end of their scopes. For instance, the scope of a local variable within a function is limited to the duration of that function. Hence, when the function returns, all its local variables are deallocated.

On the other hand, Heap Memory is used for data whose lifetime is not predetermined at compile time. This memory allocation is typically managed using the new operator in C++ and Java, or *malloc*/*calloc* in C. Deallocating data from Heap Memory can be approached in two ways: either through automatic garbage collection, as seen in Java, where the Garbage Collector autonomously manages memory, or through explicit deallocation using the *delete* operator in C++ or the *free* function in C.

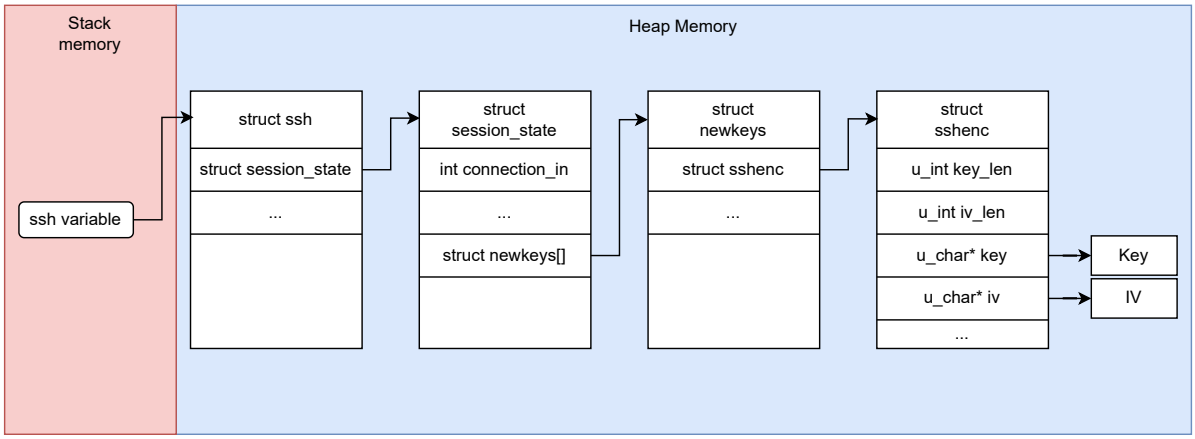


Figure 2.1: Memory layout of SSH Session Data Structures and Keys in OpenSSH

OpenSSH, which is implemented in C, allocates different memory blocks in this heap memory, to store various SSH Session data. As shown in Figure 2.1, those SSH Session data includes the hex values of the different SSH keys (Initialization Vector, Encryption Key, and Integrity Key for both client-to-server and server-to-client) used during the session [SR22; OSSHPort].

This is typically done using *malloc*/*calloc*, a standard memory allocation function in C. Consequently, capturing a snapshot of the Heap Memory at the right moment could theoretically yield a dump file containing the various SSH keys (IV, EK, IK) [Fel+22; OSSHPort].

2.3 Reinforcement Learning

Reinforcement learning (RL) is a machine learning paradigm along with Supervised and Unsupervised Learning. Reinforcement Learning does not need labeled data, it instead uses agents that explore environments by directly interacting with them. Thus, the agents learn from these past experiences. Interpreting the action it took, at which moment, along with the result of performing this action.

Reinforcement Learning is often characterized by a Markov Decision Process (MDP).

A **Markov Decision Process** (MDP) is defined by:

- S is a set of states.
- A is a set of actions.
- $P : S \times A \times S \to [0, 1]$ defines the state transition probability, with $P(s_{t+1}|s_t, a_t)$ representing the chance of moving from state s_t to s_{t+1} upon action a_t.
- $R : S \times A \to \mathbb{R}$ is the reward function, where $R(s_t, a_t)$ is the reward received after taking action a_t in state s_t.

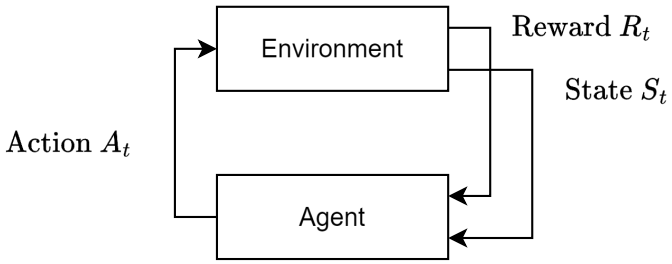


Figure 2.2: Markov Decision Process Diagram

Per this definition, at each discrete time step, denoted as t, the agent observes the current state of the environment, represented as S_t. The agent then interacts with the environment by executing an action, denoted as a_t. Subsequently, the agent observes the new state of the environment, denoted as S_{t+1}, along with receiving a reward associated with the performed action, denoted as R_t. As such, the goal of Reinforcement Learning Algorithms is to come up with a policy π that **maximizes the expected cumulative reward**.

A **policy** in reinforcement learning is formally defined as a function π, which maps a given state $s \in S$ to a probability distribution over actions $a \in A$. represented as:

$$\pi : S \to P(A)$$

Where $P(A)$ denotes the set of probability distributions over actions. For a given state $s \in S$, $\pi(s)$ yields the distribution over actions that the agent will choose from. In other words, $\pi(a|s)$ represents the probability of taking action a when in state s.

In RL, algorithms must strike a balance between two key strategies: exploration and exploitation. Exploration is when the agent chooses actions without relying on what it has already learned. This is similar to a person acting out of curiosity, trying new things to gain more information. This method is useful because it allows the agent to discover potentially better strategies that it hasn't encountered before.

Exploitation, on the other hand, is when the agent uses its existing knowledge to make decisions. Here, the agent picks actions that it believes are the best, based on its previous experiences. This approach is important because it helps the agent make the most of what it already knows.

Finding the right mix of exploration and exploitation is crucial in RL. Too much exploration can lead to poor decisions in the short term, as the agent might choose less optimal actions just to gather new information. Too much exploitation might cause the agent to miss out on discovering better strategies, affecting its long-term performance. Hence, balancing these two strategies is essential for the success of RL algorithms.

As for the classical reinforcement learning algorithms, the most well-known is probably **Q-Learning** [Wat89], which is what we call a model-free algorithm. We say that a method is **model-free** when the agent doesn't need to understand the environment dynamics, it only needs to acquire experience from the environment and focus on learning either the value of the action or the policy itself, without needing to try and model the environment's behavior. Q-Learning is value-based, which means that this method doesn't try to learn the policy directly, but rather tries to predict the *value* of taking a specific action in a specific state of a Markov Decision Process.

In the case of Q-Learning, this *value* is called the Q-Value, 'Q' represents the function that measures the **Quality** and represents the expected cumulative reward that the

agent will get by taking an action in a given state. Q-Learning works by holding a **Q-Table**, which, as its name suggests, is a table that holds the Q-Values for each possible state-action pair. The agent can then use this table to look up the Q-value of each action in a given state and choose the action with the highest Q-value.

The main idea behind Q-Learning is that the "learning" part comes from updating the Q-Values in the Q-Table, based on the rewards received by the agent after interacting with the environment.

Formally, the goal is to learn the optimal policy π^* that maximizes the expected cumulative reward. The optimal Q-Value function $Q^*(s, a)$ is defined as the maximum expected cumulative reward that can be obtained by taking action a in state s and following the optimal policy thereafter.

$$Q^*(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^*(s', a') | s, a\right]$$

Where :

- r is the reward received after taking action a in state s.
- γ, with $0 < \gamma < 1$, is the discount factor, which determines the importance of future rewards.
- s' is the next state after taking action a in state s.
- a' is the action that maximizes the Q-Value in state s'.

thus thanks to the Bellman Equation, we can update the Q-Values in the Q-Table using the following formula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$$

Where :

- α with $0 < \alpha < 1$ is the learning rate, which determines the importance of new information.
- $s, s' \in S$ are the current and next states.
- $a, a' \in A$ are the current and next actions.
- r is the reward received after taking action a in state s.
- $\max_{a'} Q(s', a')$ is the maximum Q-Value in the next state.

With this, after enough training, the Q-Table will converge to the optimal Q-Values, and we will be able to derive the optimal policy π^* by selecting the action that maximizes the Q-Value in each state.

$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

We now enter the era of **Deep Reinforcement Learning**, with Deep Q Learning [Mni+13], which enhances classical Q-Learning by substituting the traditional Q-table with a neural network as the function approximator. This adaptation allows the method to excel in complex environments with high-dimensional inputs. A typical application is in environments characterized by 'Screen State Space', where the agent perceives its surroundings solely through screen pixels. Here, the neural network processes the state inputs and outputs the Q-values for each possible action, guiding the decision-making process for action selection. Furthermore, other neural network based reinforcement learning algorithms exist, such as Actor-Critic or Proximal Policy Optimization (PPO)

To better understand this work, let us define the following notions.

An **Action Space** corresponds to the set of all possible actions that the agent can take in a particular state.

A **Discrete Action Space**, is what usually comes to mind when we speak about Action Space, here each action corresponds to one discrete action, for example, "Left", "Right", "Up" or "Down".

A **Continuous Action Space** is, as the name implies, an Action Space where each action corresponds to continuous values rather than discrete ones. Typically useful in problems such as controlling a robot arm by controlling the angle of each servo motor, where each angle would be one action.

A **Variable Action Space** is an Action Space where the number of possible actions varies between states. For example, our player could be in a state where he can no longer go "Up", which would require careful handling. This problem is usually solved by using Action Masking, which disables specific actions based on the current state in the code.

A **Fixed Action Space**, is simply an Action Space where the number of possible actions stays the same across all states.

2.4 Graphs

We now give a simple introduction to graphs, which will be the main data structure used to represent our Reinforcement Learning Environment.

A graph is a complex structure often used in computer science and mathematics to represent some sort of relationship between objects. Those objects are called nodes or vertices, and the relationships between them are called edges. The edges can be either directed or undirected, depending on the nature of the relationship they represent. If an edge is directed, it means that the relationship between the two nodes is one-way. If an edge is undirected, it means that the relationship between the two nodes is bidirectional.

We say that a graph is **directed** if all its edges are directed, similarly, we say that a graph is **undirected** if all its edges are undirected.

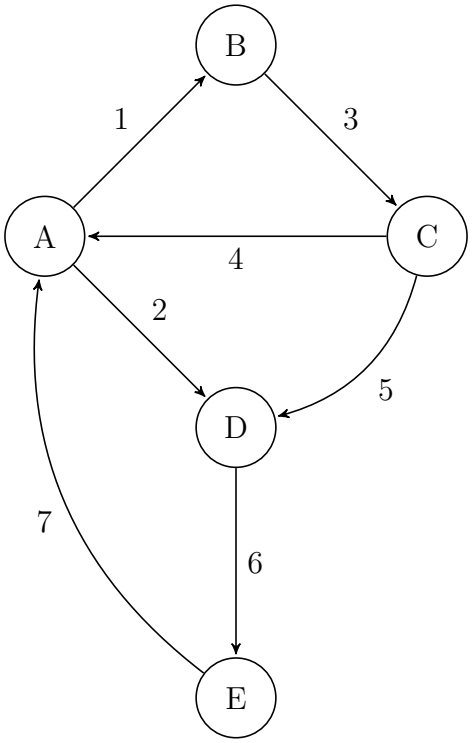


Figure 2.3: Example of a directed graph

Edges can also have weights, which represent the strength of the relationship between the two nodes. Therefore, a **weighted graph** is a graph where each edge has a weight.

Two vertices are said to be **adjacent** if there is an edge between them. For example in Figure 2.3, the vertices A and B are adjacent, as there is an edge between them.

The **degree** of a vertex is the number of edges incident to it. We can also define the **in-degree** and **out-degree** of a vertex in a directed graph, which are respectively the number of incoming and outgoing edges of the vertex. For example in Figure 2.3, the vertex A has a degree of 4, an in-degree of 2, and an out-degree of 2.

A **path** in a graph is a sequence of vertices such that each consecutive pair of vertices is connected by an edge. We can also note that in a directed graph, a path is only valid if the edges are directed in the right way. For example in Figure 2.3, the sequence of vertices A, B, and C is a path.

A **cycle** is a path that starts and ends at the same vertex. For example in Figure 2.3, the sequence of vertices A, B, C, A is a cycle.

2.5 Graph Neural Networks

Graph Neural Networks (GNNs) [GNN] are a class of neural networks that can operate on graph-structured data. They are becoming more and more popular, as they are effective in a wide variety of problems implying graph-structured data, for instance, they are commonly used in Graph Classification, Node Classification, and Edge Prediction. One recent use case of such architecture is the GraphCast model from Google [Lam+22], which used GNNs for weather forecasting and managed to compete with or even outperform some industry-standard weather prediction models in terms of accuracy and speed. It became the state of the art and they even made the model architecture open source and are also making the weights available. Effectively making anyone with enough computing power to have their own weather forecasting system.

There are multiple types of Graph Neural Networks, like Graph Convolutional Networks, Graph Auto Encoder Networks, or even Graph Recurrent Networks. The most common one is the Graph Convolutional Network, which is the one we will use in this work.

A core concept of GNNs is **Message Passing**, which is a mechanism where the nodes of the graph exchange information with their neighbors and aggregate this information

to update their features. It allows the nodes embedding to capture both structural information of the graph and the features of the nodes themselves.

The Graph Convolutional Network (GCN) is a type of GNN that uses Graph Convolutional Layers that utilize this message-passing mechanism to update the node embeddings. Similarly to regular Convolutional Layers, the Graph Convolutional Layers often use some sort of weighted sum of the neighbors' features to update the node's embedding.

There exist multiple variants of Graph Convolutional Layers, like the GCN [KW16], GraphSAGE [HYL17], or even the GAT [Vel+17]. Each of these variants has its specific way of aggregating the neighbors' features and updating the node embeddings.

3 Related Work

We now turn our attention to the existing literature that forms the foundation of our research. Initially, we delve into the field of Digital Forensics, with a particular focus on the extraction of encryption keys from remote communications. Following this, we explore the domain of Deep Reinforcement Learning, a technique that has found applications across various fields, including cybersecurity. This exploration enhances our understanding of the problem at hand and underscores the motivation behind our proposed method.

3.1 Digital Forensics

In the field of Digital Forensics, several studies have addressed the challenge of recovering encryption keys to decrypt remote communications.

In 2016 Taubann et al. published TLSKex [Tau+16]. The work explores the extraction of session keys from TLS communications using Virtual Machine Introspection (VMI). VMI serves as a means to monitor and analyze the state of a virtual machine (VM) from an external viewpoint, allowing the inspection of a VM's operating system and applications without interfering with its operation. This methodology offers valuable insights into the extraction of SSH Session keys.

The study outlines strategies for capturing TLS communications via either an external device or software embedded within the VM. It emphasizes the critical importance of timely memory acquisition to guarantee the availability of all session keys. Moreover, it stresses the need for rapid execution of this process to minimize disruption to network communication, thus ensuring a seamless and unnoticeable integration. Additionally, the research addresses key extraction via brute force, proposing strategies to simplify this intensive process by reducing the size of the problem.

In 2018, Taubann et al. also published a paper [TAR18], aiming at efficiently extracting TLS session keys. This work gives us some observation on the memory layout, and how the process data is stored, also focusing on effectively timing the extraction process to be handled in real-time.

Similar to TLSKex [Tau+16], Sentanoe et al. published a work called SSHKex [SR22]. They aimed at decrypting SSH network traffic this time, by extracting SSH Session Keys from the main memory using VMI. They take advantage of the known data structure of the Open SSH implementation, to follow the path to the SSH Session Keys and other metadata. They highlight the inner technical workings of the SSH protocol and the OpenSSH implementation., On top of that, they provide an easy-to-use, plugin for WireShark [Wireshark] (A network protocol analyzer) and a Python script that decrypts the SSH communication by giving them the extracted session keys. The main limitations of this method are that it requires a priori knowledge of the OpenSSH implementation and that it needs to pause the execution flow of the OpenSSH process to extract the keys. Those limitations make it hardly generalizable to other SSH versions and implementations and also introduce some potentially detectable artifacts in the system.

Fellicious et al.[Fel+22] proposed a work that utilizes Machine Learning to extract the SSH session keys directly from the Heap Dump of an OpenSSH Process. They improved SSHKex by adding to it the ability to automatically dump the heap memory of the OpenSSH process, and support for SSH client monitoring. They used this improved version of SSHKex to generate a complete Dataset, that contains the raw heap dump as well as a corresponding JSON that contains all the important structures, such as the keys, as well as their address within the heap dump file. They used a Random Forest classifier, some well-crafted features as well as techniques to reduce the size of the heap dump as much as possible, to predict which entry in the heap dump is a key, and which is not. These methods had the advantage of being able to launch the decryption process even after the OpenSSH process has been terminated, and on another machine, which is a big advantage in terms of stealthiness. However, the main limitation of this method is that it requires to be trained on a substantial amount of data, thus making it hard to use it on new versions of OpenSSH, or even on different implementations.

The insights gained from these prior studies provide valuable assets that we can apply directly to our problem. In particular, they help us develop a nuanced understanding of the SSH protocol and key negotiation processes. This includes how certain SSH implementations, such as OpenSSH, store keys, as well as clarifying the objectives and

motivations behind employing such methods. This foundational knowledge sets the stage for our exploration into the application of Deep Reinforcement Learning in similar contexts.

3.2 Deep Reinforcement Learning

Regarding Deep Reinforcement Learning, a substantial amount of research has established a solid foundation in the field.

Google Deep Mind moved Deep Reinforcement Learning forward, by providing numerous works. One of the pioneer papers is published in 2013 [Mni+13]. This work was the one that first introduced Deep Q-Netowrks, effectively showing the effectiveness of mixing Deep Neural Networks and Q-Learning, by teaching an agent to play ATARI Games better than most (if not any) humans. It also highlights the capacity of Deep Reinforcement Learning to generalize on a multitude of different environments and to learn from raw pixel data, which is a big advantage in terms of generalization capabilities.

We also have the very famous ones from Google DeepMind, like AlphaGO [Sil+16], which marked the beginning of a new era, as it was the first time a software managed to beat one of the best players of a name called GO, which is a strategy game with a vast possible overall play space. It did this by utilizing a method called Monte Carlo Tree Search (MCTS) [Świ+21] and Deep Neural Networks, learning not only from human examples but also by playing against itself, effectively discovering new and efficient, moves and strategies.

Another significant contribution was made by Lillicrap et al. [Lil+15], who addressed continuous control challenges within deep reinforcement learning. They proposed a model called Deep Deterministic Policy Gradient (DDPG), which employs an Actor-Critic architecture. Unlike Deep Q-Learning, which is limited to relatively small and discrete action spaces, DDPG can effectively operate within large and continuous action spaces. This capability is critical for handling more complex and varied control tasks.

Additionally, Actor-Critic algorithms [Mni+16] employ a dual-model strategy to navigate complex environments efficiently. The 'Actor' decides on actions to take from a given state, utilizing either stochastic or deterministic methods, while the 'Critic', similarly to Deep Q-Learning, evaluates the potential value (expected return) of actions from

a state or the desirability of the state itself, independent of specific actions. By exchanging feedback during training, this collaboration enhances learning, proving particularly effective in complex scenarios.

Furthermore, some works [Kor24; Mni+13] have highlighted the generalization capabilities of Deep Reinforcement Learning methods. Generalization capabilities are crucial for certain problems, especially when applying our methods to real-world scenarios. This allows our agent policy to perform well even in unseen environments, enhancing its overall robustness.

A survey by Korkmaz [Kor24] summarizes the significant successes of deep reinforcement learning (DRL) in various domains. It addresses challenges in generalization and robustness due to overfitting in state-action value functions and explores solutions to enhance these aspects in DRL policies. The primary reasons for a policy's limited robustness and generalization include insufficient exploration of the high-dimensional state space in our Markov Decision Problem (MDP) and the inherent limitations of using Neural Networks as approximators.

These works have been particularly helpful in designing our methods, especially in defining the exploration strategy and the training function. It also comforted us in the choice of using Deep Reinforcement Learning for our method, as it seems to be able to generalize well in different environments, thus highlighting the capacity to learn from relatively small amounts of data.

In this chapter, we have explored the existing literature in the fields of Digital Forensics and Deep Reinforcement Learning. We highlighted the strengths and limitations of the existing methods, and the potential benefits of using Deep Reinforcement Learning in the context of Digital Forensics. Our method aims at improving the capacity of extracting SSH Session Keys with the least amount of intrusion possible, and with the ability to perform well even when trained on limited data by leveraging the generalization capabilities of Deep Reinforcement Learning.

4 Methods

In this chapter, we delve into the more technical aspects of this work. In particular, we take a look at the dataset used as well as the different tools, models, and algorithms used.

4.1 Dataset

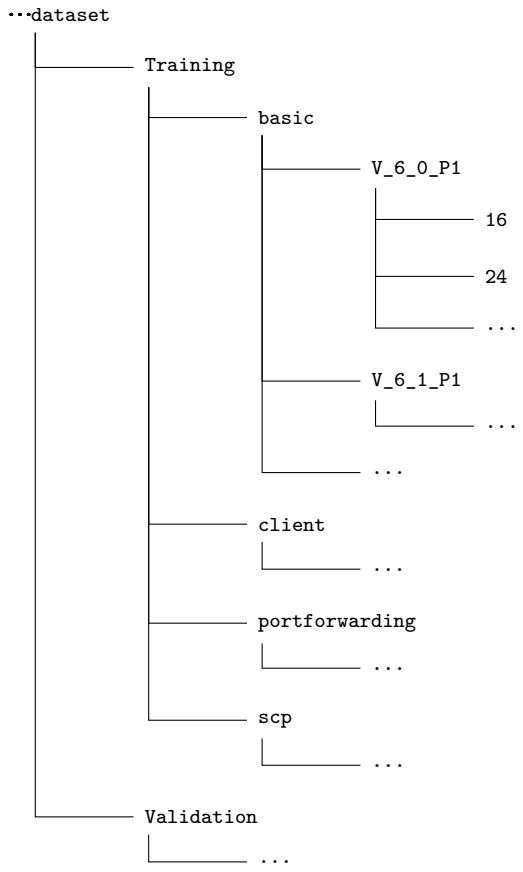


Figure 4.1: Tree structure of the dataset directories

For this work, we used a dataset generated by Fellicious et al. [Fel+22]. This dataset includes the RAW files containing the heap dump of an OpenSSH process, accompanied by their corresponding JSON files with metadata about the process and the heap dump (as shown in Figure 4.2). Specifically, the fields that are of interest to us are :

- `HEAP_START`: The address of the beginning of the heap in the memory.
- `KEY_A, KEY_B, …, KEY_F`: These fields contain the hex values of the keys used in the SSH communication.
- `KEY_A_ADDR, KEY_B_ADDR, …, KEY_F_ADDR`: The address of the keys in the memory in hex.

Furthermore, all the addresses present in the JSON file are the actual addresses in the virtual memory. This means that to access the data at this address within the heap dump file, we need to subtract the `HEAP_START` address from the address in the JSON file (as shown with Figure 4.3).


```
{
  "SSH_PID": "26518",
  "..."
  "KEY_A_ADDR": "55dfa5c79170",
  "KEY_A_LEN": "16",
  "KEY_A_REAL_LEN": "16",
  "KEY_A": "7a89bf37e132567112a700e659005b79",
  "..."
  "HEAP_START": "55dfa5c6a000",
}
```


Figure 4.2: Snippet of the Metadata JSON file

```
1  Address : 16 bytes value
2  0000f160: 3000 0000 0000 0000 3100 0000 0000 0000 0.......1.......
3  0000f170: 7a89 bf37 e132 5671 12a7 00e6 5900 5b79 z..7.2Vq....Y.[y
4  0000f180: 659e 7002 2be0 d006 ee22 d3cb ee7c 613b e.p.+....".../a;
```

Figure 4.3: Snippet of the RAW heap dump file showing KEY_A

As shown in Figure 4.1, the dataset is organized into two main directories: *Training* and *Validation*. Each directory is further subdivided by SSH scenario (*basic*, *client*, *port-forwarding*, *scp*), then by different versions of OpenSSH, and finally by the key length used in the SSH communications. Note that for some versions of OpenSSH, the tool was

only capable of extracting the encryption key values (for versions before V_6_8_P1), which we decided to discard from the dataset.

4.2 Method Outline

Let us first outline the method we are proposing in this work.

In this work, we propose a method that utilizes Deep Reinforcement Learning to teach an agent to navigate through a Graph generated from a heap dump file and identify the nodes that contain the keys used in the SSH communication.

As outlined in Figure 4.4, the initial step involves generating a Pointer Graph from the heap dump file. We use this graph as the environment for our Reinforcement Learning Agent to explore. Multiple factors motivated the choice of this data structure :

- It allows us to represent the heap dump in a more structured way, making it easier for the agent to navigate through it.
- By design, the graph representation filters out most of the noise and irrelevant data, thereby reducing the problem's complexity.
- Pointer Graphs are particularly relevant in the semantic analysis of memory dumps. They are perfectly suited for identifying specific structures within these dumps, as these structures manifest as subgraphs within the Pointer Graph. This capability could be useful for accurately mapping and understanding the complex semantic relationships in memory layouts.

For this purpose, we require a program specifically designed to process the raw heap dump files and the associated JSON data files. This program will output a graph where each node represents a pointer. The edges between these nodes depict the connections among these pointers. Additionally, each node is enriched with features, and labeled to indicate whether the pointer corresponds to a key. If it does correspond to a key, the label further specifies **which** key (Key_A, Key_B, Key_C, Key_D, Key_E or Key_F) it is associated with (See section 4.6).

It is important to note that the JSON data file, while essential for generating both "Training" graphs and graphs used in "Production," differs in its content depending on

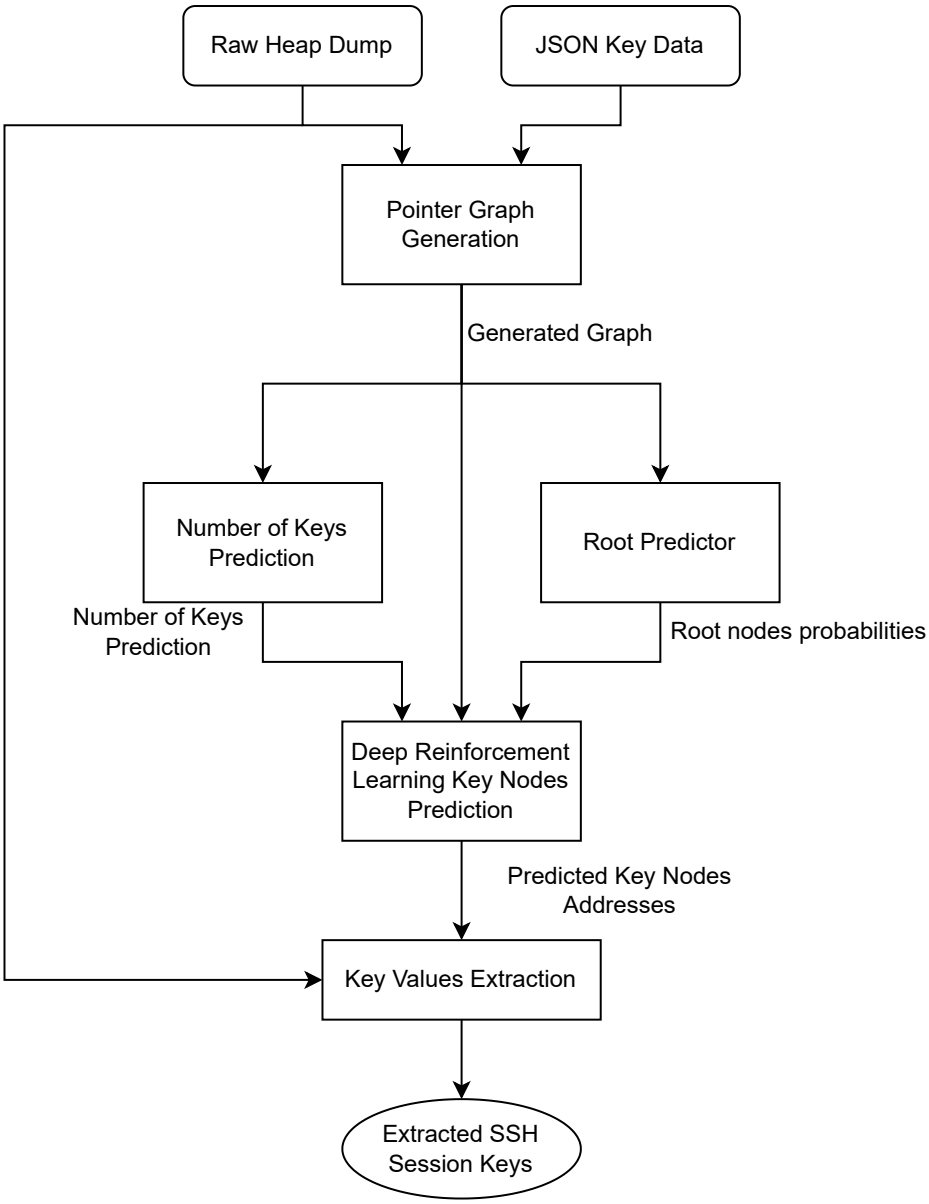


Figure 4.4: Flow Diagram of our method

the use case. In "Production," unlike in "Training," the JSON file does not contain information about the keys, as the primary goal of this work is for the agent to independently discover these keys. However, the file still provides crucial data necessary for graph creation, such as the starting address of the heap. For a comprehensive discussion on the graph generation process, see section 4.4; for details on the dataset structure, refer to section 4.1.

This graph is then passed to a classifier designed to predict the number of keys present within the graph. Beyond simply guiding the agent in determining the number of keys it needs to locate, we will demonstrate how from this predicted number of keys, it is possible to infer **which** specific keys are present. This ability allows the agent to focus on the relevant keys, thus reducing the complexity of the problem.

We also input the graph to another classifier, this time designed to predict the initial root node, from which the agent will start exploring the graph. The aim here is also to simplify the task of the agent, by providing it with a starting point that is likely to lead to the keys.

Finally, we will train a Deep Reinforcement Learning Agent to navigate the Pointer Graph and accurately identify the nodes that contain the keys. The training process will utilize the Pointer Graph as the environment, incorporating the predicted root node and the number of keys as critical inputs. We then use those identified key nodes to extract the actual key values from the heap dump.

With the methodological foundation set, we will next explore the specific tools and frameworks that enable the implementation of these processes.

4.3 Tools and frameworks used

In this section, we will discuss the different tools and frameworks used in this work.

For most of our work, we decided to go with Python [Python], as it is a common choice for deep-learning-related tasks. It also benefits from a large supportive community and extensive learning resources, which facilitate faster learning and troubleshooting. However, for the graph generation component, we opted to use Rust [Rust]. This choice

was driven by Rust's well-known performance advantages, particularly necessary for processing large files.

Choosing the right Deep Learning Framework is an essential decision in any deep learning project. Several options were considered, including TensorFlow [TensorFlow], Keras [Keras] (which operates atop TensorFlow), and PyTorch [PyTorch]. We ultimately chose PyTorch due to its balance of abstraction and low-level control, which facilitates a manageable learning curve while still allowing for detailed, in-depth customization when necessary. Additionally, PyTorch is supported by comprehensive documentation [PyTorchDoc] and a robust suite of tools and libraries, such as PyTorch Geometric [PyG], which are invaluable for our tasks.

PyTorch Geometric [PyG] (PyG for short), is a library that works on top of PyTorch that allows it to handle Graphs. It implements some efficient data structure, data loaders, utils functions, Graph Convolutions, Graph Pooling layers as well as other features. It also has very well crafted documentation [PyGDoc], and a lot of examples that can help us in our task.

For handling the graph in a more general manner, we used the NetworkX library [NetworkX]. It provides an easy way of accessing and altering some specific graph features and has some, useful, out-of-the-box utility algorithms like BFS, DFS, and many other graph-related operations.

For the Root Node Classifier in section 4.7, we use scikit-learn [sklearn], a simple and efficient tool for data mining and data analysis, built on NumPy, SciPy, and matplotlib.

4.4 Graph Generation

In this section, we go through the process of generating the Graph from the heap dump file that is used as the environment for the Reinforcement Learning Agent. Throughout this work, the graph can be referred to as Pointer Graph, Heap Graph, Generated Graph, or simply Graph.

This graph represents a network of pointers, where each node corresponds to a memory block allocated by the `malloc` function, and each edge indicates a pointer from one memory block to another. We define the Pointer Graph formally as a directed graph $G_{\text{heap}} = (V, E)$, where:

- V is the set of nodes, with each node $v \in V$ representing a memory block allocated by the `malloc` function.
- E is the set of edges, where an edge (u, v) represents a pointer from the block represented by node u to the block represented by node v, for all $u, v \in V$.

The main idea behind this graph G_{heap} is to represent the heap dump in a more structured way, making it easier for the agent to navigate through it.

Moreover, the heap dump, which was captured from the virtual memory, consists of virtual addresses. This means that each address contained in the dump doesn't correspond directly to locations in the heap **dump file**. This means that to access the data at this address within the heap dump file, we need to subtract the `HEAP_START` address from the corresponding addresses in the virtual address space. This lets us know the address relative to the heap dump file, which is necessary to access the data within it. We illustrate this concept with the example from section 4.1 in Figure 4.5.

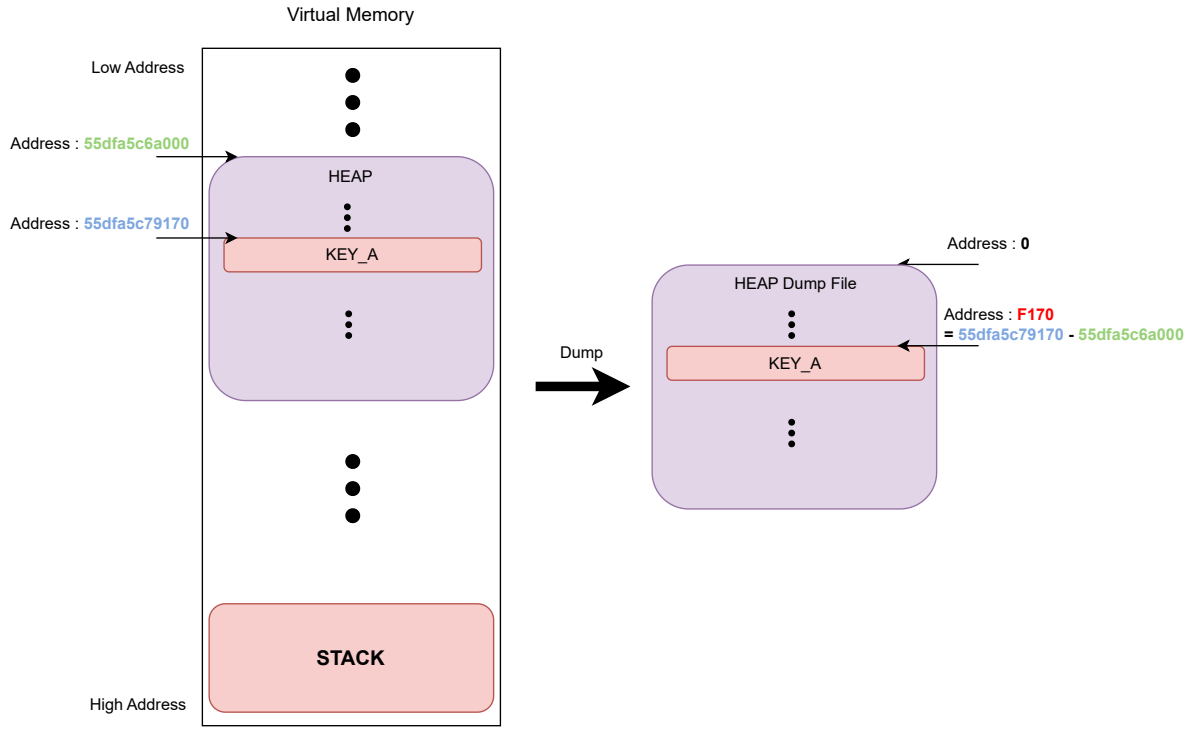


Figure 4.5: Example of conversion from Virtual Address to Heap Dump Address

From there we can begin the process of generating the graph. Our algorithm is similar to a Depth-First Search (DFS) algorithm. It involves having a set $\mathcal{P}$ of all the potential

pointers within the heap dump file and then iterating over it. For detecting those pointers, we use a simple regex that matches the hex format of a pointer. We then iterate over all the potential pointers and create the graph by recursively processing the pointers within the block allocated by the `malloc` [Malloc] function. Furthermore, we can simply add the $offset$ attribute to each edge created between the 2 blocks. A high-level overview of the graph generation algorithm is provided in Figure 4.6. Also, note that the $struct\_size$ used to define the search area, is obtained from the `malloc header` [Malloc; MallocGit], which is located 8 bytes before the pointed address. Moreover, to get the size of the **payload** of the block, we have to subtract the size of the `malloc header` from the $struct\_size$ [MallocGit].

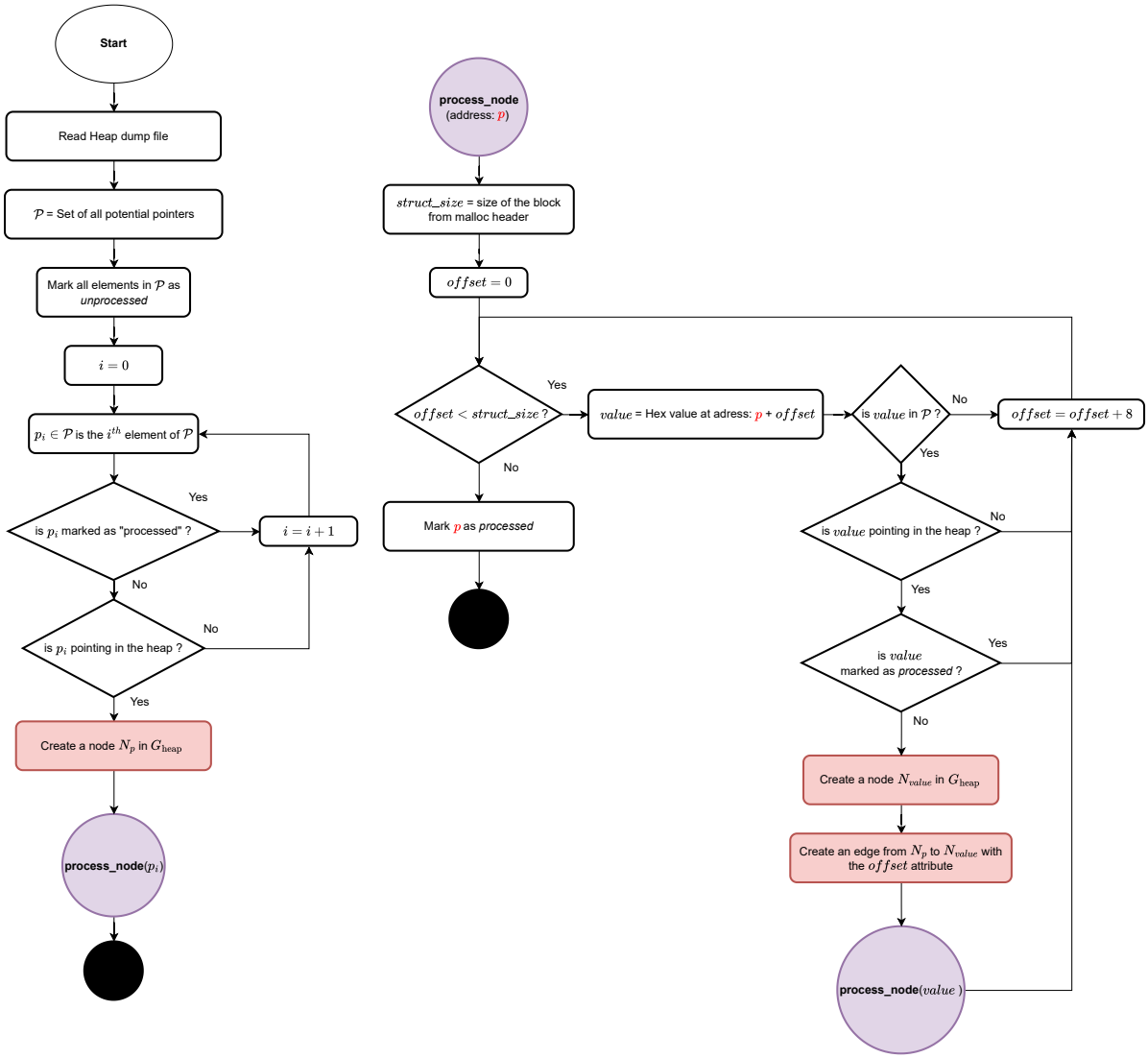


Figure 4.6: High-Level view of the Graph Generation Algorithm

Moreover, for each node that is created, we add the following features:

- $struct\_size$: Size of the corresponding malloc block
- $address$: The address of this node in the global memory space
- $valid\_pointer\_count$: The number of valid pointers within the block
- $invalid\_pointer\_count$: The number of invalid pointers within the block
- $first\_pointer\_offset$: The offset of the first found pointer within the block
- $last\_pointer\_offset$: The offset of the last found pointer within the block
- $first\_valid\_pointer\_offset$: The offset of the first found valid pointer within the block
- $last\_valid\_pointer\_offset$: The offset of the last found valid pointer within the block
- cat : The index of the corresponding Key if found in the JSON.

We assign the *cat* feature based on a key that matches the address of the node in the JSON file. For this category attribute to be easily processed later on, we convert the keys to a numerical value :

- No Key: $cat = 0$
- Key A: $cat = 1$
- Key B: $cat = 2$
- ...
- Key F: $cat = 6$

This *cat* feature is used as a label for our different methods. Note that this feature will, of course, be automatically set to -1 for all nodes in real-world testing scenarios (as there won't be any match in the JSON). The process used to generate the attributes can be seen in Figure 4.7.

Figure 4.8 shows an example of a generated Pointer Graph. The graph has been generated from a heap dump of OpenSSH Basic V_7_9_P1 with four 32-bit keys. We can see that the graph is large, with **839** nodes and **921** edges. Note the presence of isolated nodes and islands, as well as the clear difference in degree between nodes. This also demonstrates the complexity of finding the keys (represented in light colors in the figure).

After the graph is generated, we save it in a .graphml file [GraphML], which is an XML-based file format, that allows us to save the graph structure, as well as the node

and edge attributes. This file can then be loaded into NetworkX and converted to a PyTorch Geometric Data object, which is the format used by PyTorch Geometric to handle graphs. However, while .graphml is technically able to store multiple feature formats, the Rust library we used to generate them, forced us to export them as strings. Thus a preprocessing step is needed to convert them to the correct format.

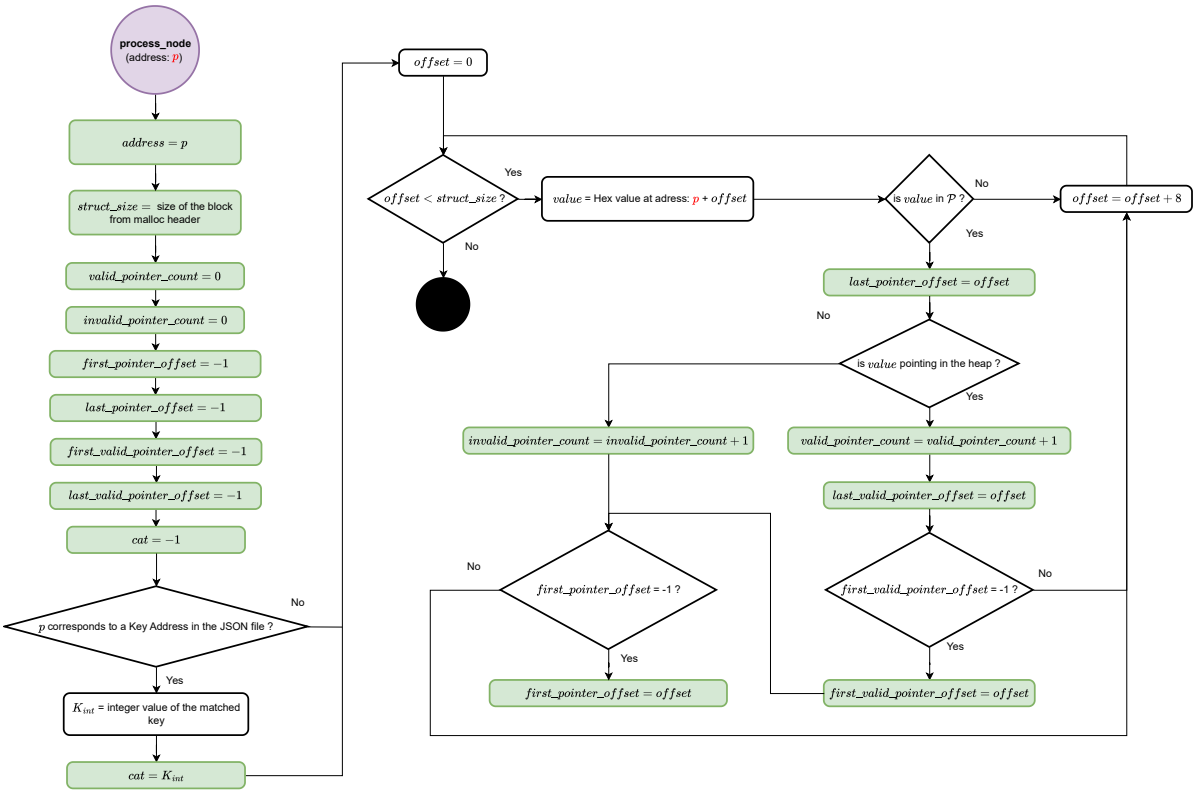


Figure 4.7: High-Level view of the node features creation

4.5 Graph Preprocessing

The graph that we generated from the heap dump is quite large and is not very well suited to any Machine Learning tasks. As such, in this section, we go over the preprocessing step needed to be able to pass those as input to our different models.

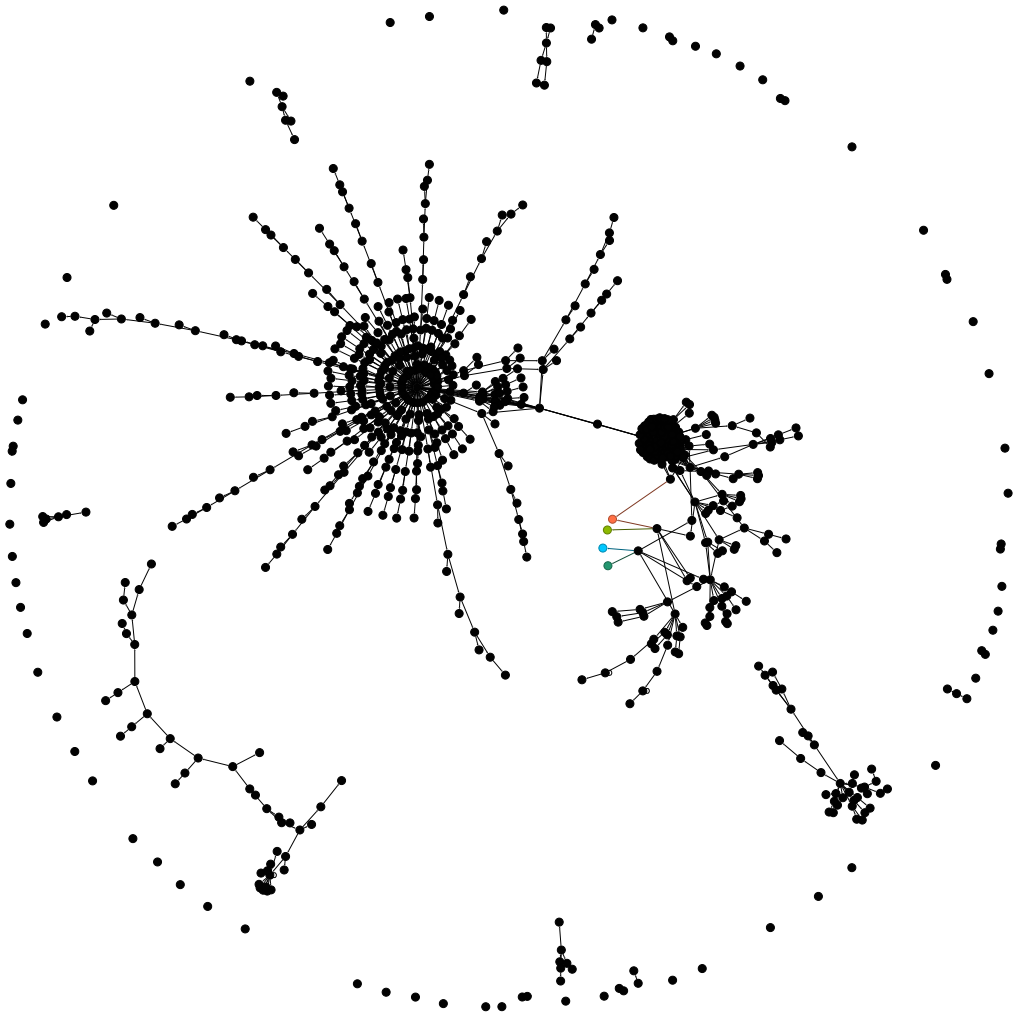


Figure 4.8: Example of a Pointer Graph generated from a heap dump

4.5.1 Global Preprocessing

As mentioned in section 4.4, the exported .graphml file contains the node and edge features as strings. Thus we first need to convert those to the appropriate format.

This process is straightforward as all of the learnable features can be represented as integers. For this we load the .graphml file using NetworkX, and then iterate over all the nodes and edges, converting the features to integers. The only field that is kept as a string is the *address*, as it is not used as input to our models, and only serves as a reference to the original memory address.

Once this is done, we can then further preprocess our graphs according to the method we are using, note that at this point the graph is represented as a NetworkX Graph object.

4.5.2 Preprocessing for PyTorch Geomtric

To convert our graphs to a suited input for our Deep Learning models, we need to convert them to PyTorch Geometric Data objects [PyGData]. This object is a simple data structure that contains the graph sparsely. In our case, the Data structure contains the following attributes :

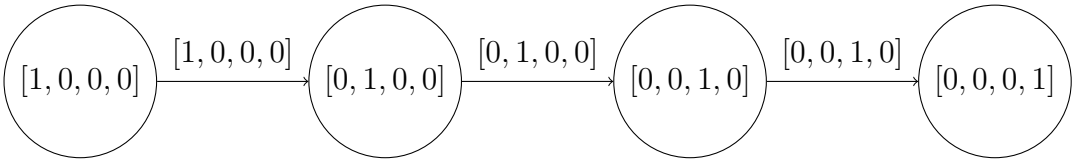
- x : The node features matrix
- $edge\_index$: The edge index matrix
- $edge\_attr$: The edge features matrix

We can also note that this Data object can hold other attributes, such as the y attribute, which would be the target value for a classification task.

To convert our graph from a NetworkX graph object to a PyTorch Geometric Data object, two ways are possible :

- Using the PyTorch Geometric library to convert the NetworkX graph object to a PyTorch Geometric Data
- Manually converting the NetworkX graph object to a PyTorch Geometric Data

The first method is the "easiest", as it only requires a single line of code. However, we had some issues with this method, as it required the indices of the graph to be in a numerical format, with contiguous numbers. This was not the case for our graphs, as such we decided to manually convert the NetworkX graph object to a PyTorch Geometric Data object. This method while more complex also gives us more flexibility on which features we want to keep or discard. For this, we had to create a mapping between the real node indices of the NetworkX graph object and the contiguous indices of the PyTorch Geometric Data object. The main reason for this is that the PyTorch Geometric Data object requires PyTorch tensors, which are indexed by contiguous integers. For an example of how a Data Object could be constructed, consider this directed graph :



There are 4 nodes, with the following features :

- Node 0 : $[1, 0, 0, 0]$
- Node 1 : $[0, 1, 0, 0]$
- Node 2 : $[0, 0, 1, 0]$
- Node 3 : $[0, 0, 0, 1]$

And the following edges :

- Edge 0-1 : $[1, 0, 0, 0]$
- Edge 1-2 : $[0, 1, 0, 0]$
- Edge 2-3 : $[0, 0, 1, 0]$

We can then convert this graph to the following PyTorch Geometric Data object :

- x : $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- $edge\_index$: $\begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 \end{bmatrix}$

- *edge_attr* : $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

4.5.3 Graph Key Classifier

For the Graph Key Classifier in section 4.6, we further preprocess our PyTorch Geometric Data object. By standardizing the features at the graph level, we ensure that the features have a mean of 0 and a standard deviation of 1. Which made the training process faster and more stable. We standardized both the node features and edge features independently.

Formally :

$$z_i = \frac{x_i - \mu}{\sigma}$$

Where :

- z_i is the standardized features of node i in the graph
- x_i is the original features of node i in the graph
- μ is the mean for each feature for the whole graph
- σ is the standard deviation of each feature for the whole graph

The same process is applied to the edge features.

4.5.4 Deep Reinforcement Learning

For the Deep Reinforcement Learning part in section 4.8, we only remove the isolated nodes from the NetworkX graph object, before converting it to a PyTorch Geometric Data object. The goal here is to simplify the task of the agent, by removing nodes that can't be Key nodes, as we assume that should be part of a structure that contains the keys.

4.5.5 Root Node Classifier

For the Root Node Classifier in section 4.7, no further preprocessing is needed. The NetworkX Graph object is passed as input to the program as is.

4.6 Graph Keys Classifier

In this section, we discuss the design and implementation of the Graph Keys Classifier.

The main motivation behind the creation of this classifier is to ”simplify” the task of the agent. Based on the OpenSSH implementation and versions used, the keys utilized for the SSH communication might vary. We won't go into too much detail, but this is mainly due to the different encryption algorithms used, which require different keys [YL06].

As explained in subsection 2.1.1 :

- Key_A and Key_B are the initialization vectors and thus are sometimes needed to encrypt and decrypt the data.
- Key_C and Key_D are the encryption keys and are always needed to encrypt and decrypt the data.
- Key_E and Key_F are only used to check the integrity of the data and, thus are not necessary when encrypting and decrypting the data.

So for this work, we decided that the agent would only focus on the keys that are present in the input graph. Avoiding adding another layer of complexity to the agent, where it would have to learn which keys to explore, and when to stop searching for other keys. We do this by outsourcing this problem to another Neural Network classifier, that will try and predict how many keys are present within the graph, and then try to infer which ones are present in the graph generated from the heap dump.

We know that those keys usually work in pairs (client->server) and (server->client), thus we can assume that the graph would contain either 2, 4, or 6 keys. However, defining which keys are being used only from the number of keys requires some thought. First of all, for every encryption algorithm, we always need at least Key_C and Key_D,

as they are the encryption keys. From that, we can already infer that if the classifier predicts 2 keys, those must be Key_C and Key_D. Then if the classifier predicts 6 keys, easily we know that every key is present.

Finally, the hard part comes when the classifier predicts 4 keys, where we have the choice between the initialization vectors and the integrity keys. To address this challenge, we developed a script that processes our graph data from the training dataset. This script is designed to ascertain whether integrity keys or initialization vectors are detected, whenever exactly four keys are identified. We found out that in our training dataset, whenever there are 4 keys in the graph, those are always Key_A, Key_B, Key_C, and Key_D. For that reason, we assume that if the classifier predicts 4 keys, those must be Key_A, Key_B, Key_C, and Key_D. We have tried to take a look at the OpenSSH source code [OpenSSH] as well as the RFCs [YL06] to try and find out why that was the case, but we didn't manage to find any information on this, mainly due by the complexity of the source code.

Thus, from an input graph, the classifier will output one of those classes :

- Class 0: 2 Keys have been detected, must be Key_C and Key_D
- Class 1: 4 Keys have been detected, must be Key_A, Key_B, Key_C, and Key_D
- Class 2: 6 Keys have been detected, must be Key_A, Key_B, Key_C, Key_D, Key_E, and Key_F

As far as the model architecture goes, we used a "simple" Graph Neural Network design, using the GATv2Conv layer from the PyTorch-Geomtric library.

According to the official Pytorch Geometric Documentation [Gv2Conv], the GATv2 Conv layer works by aggregating the features of all neighbors of each node in the graph in this same node, by learning an attention factor for every neighbor, it can also take the edge features into account in its attention calculation.

Formally :

$$\mathbf{x}'_i = \alpha_{i,i}\mathbf{\Theta}_s\mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}_t\mathbf{x}_j,$$

with the attention coefficient being calculated by :

$$\alpha_{i,j} = \frac{\exp\left(\mathbf{a}^\top \text{LeakyReLU}\left(\mathbf{\Theta}_s \mathbf{x}_i + \mathbf{\Theta}_t \mathbf{x}_j + \mathbf{\Theta}_e \mathbf{e}_{i,j}\right)\right)}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp\left(\mathbf{a}^\top \text{LeakyReLU}\left(\mathbf{\Theta}_s \mathbf{x}_i + \mathbf{\Theta}_t \mathbf{x}_k + \mathbf{\Theta}_e \mathbf{e}_{i,k}\right)\right)}.$$

Where :

- $\alpha_{i,j}$ is the attention coefficient that measures the importance of node j's features to node i.
- $\mathbf{x}_i$ and $\mathbf{x}_j$ are the feature vectors of nodes i and j, respectively.
- $\mathbf{e}_{i,j}$ is the edge feature vector from node i to node j.
- $\mathbf{\Theta}_s$, $\mathbf{\Theta}_t$, and $\mathbf{\Theta}_e$ are learned weight matrices for the source node i, target node j, and edge $e_{i,j}$, respectively.
- $\mathbf{a}$ is a learned weight vector that makes the attention mechanism adaptive.
- $\mathcal{N}(i)$ denotes the set of neighbor nodes of i, including i itself.

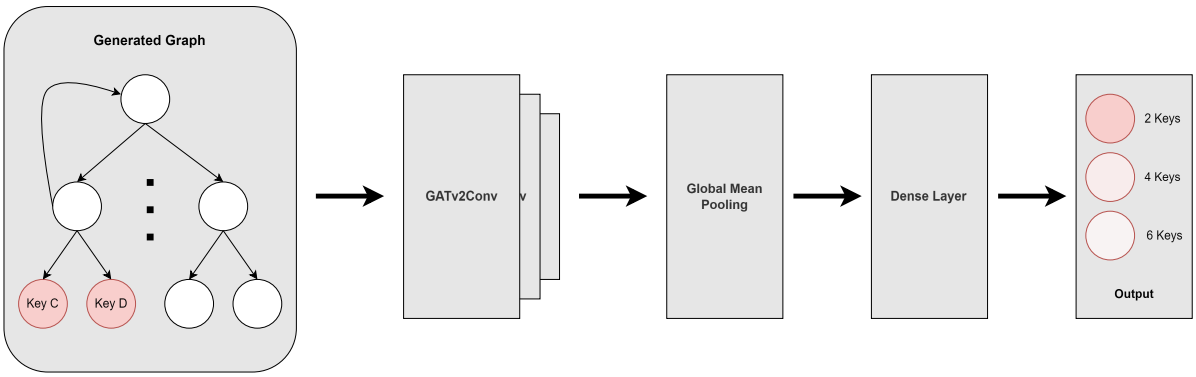


Figure 4.9: Key Classifier Neural Network Architecture

4.7 Initial root prediction

In this section, we discuss the motivation and design of the Initial Root Prediction Classifier.

Ideally, we would have liked our environment to be the entire generated graph from the raw heap dump file. Where we would train the agent to start at any node, and then learn how to navigate this graph node by node, through existing edges to find the keys. However, As Figure 4.8 perfectly illustrates, the generated graph is comprised of a lot of

disconnected subgraphs. This means that we can't let the agent start from any random node, as it might not be able to reach the keys.

Moreover, we can't just have a model that predicts which subgraph might contain the keys, and then randomly choose a starting node from this subgraph. This wouldn't be an issue if our graphs were undirected, however, our graphs are directed. This means that there may be nodes within the same subgraph that don't have any paths to the key nodes. This scenario is illustrated in Figure 4.10.

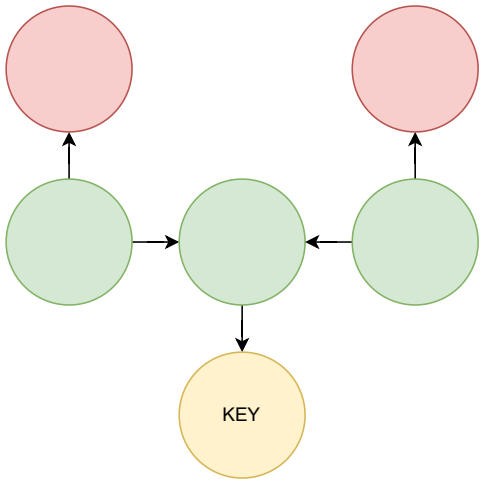


Figure 4.10: Example of a directed subgraph with valid (green) and invalid (red) starting nodes

From those characteristics, we can see that the choice of the root node is crucial. Initially, we defined $\mathcal{R}$ the set of potential root nodes such that $\mathcal{R} = \{v \in V \mid \text{indegree}(v) = 0\}$. Thus we define potential root nodes as nodes that don't have any incoming edges.

Subsequently, our idea was to simply create a Master Node, that would be connected to all the nodes in $\mathcal{R}$. But this would typically lead to this node having an outdegree that is 50 to 100 times more than all the other nodes. This can be a problem in Deep Reinforcement Learning, as it increases the model complexity, and implies paying particular attention to the design of the action space. Thus we quickly realized that while being the "easiest" to implement, it is not the most efficient way to solve this problem.

After that, we made a discovery that made us reconsider how we define potential root nodes. Depending on the version of OpenSSH used, the structure used to store the keys might change. We discovered that this structure not only holds the keys but may

also include pointers to other structures. These, in turn, point back to the original key-containing structure, thus forming a cyclical relationship. This makes choosing the root **only** from nodes without any parents infeasible. This scenario is illustrated in Figure 4.11.

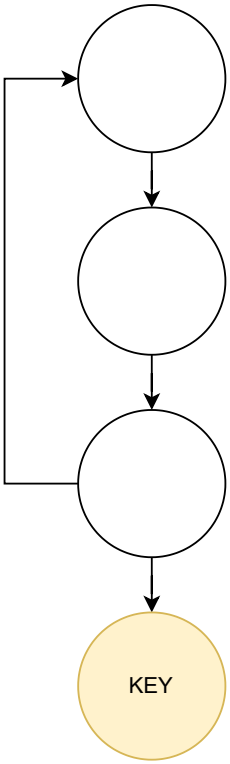


Figure 4.11: Example of a directed subgraph with no valid root

So to remedy this problem, we utilize a famous concept of Graph Theory called **Strongly Connected Components**. We say that a Directed Graph is strongly connected if every node within this graph can reach every other node. Thus Strongly Connected Components are subgraphs in a Directed Graph that are Strongly Connected.

Each SCC in G is abstracted as a single *meta node* in a new graph G'. This abstraction simplifies G by treating each SCC as a node in G', with edges between meta nodes that correspond to direct connections between different SCCs in G.

A *root meta node* in G' is a meta node with no incoming edges. The set of all vertices in G that belong to an SCC represented by a root meta node in G' is denoted by $\mathcal{R}$. Thus,

$$\mathcal{R} = \{v \in V \mid \text{SCC}(v) \text{ has no incoming edges in } G'\}$$

where SCC(v) is the strongly connected component containing v.

This definition bypasses the issue of cyclical relationships, as $\mathcal{R}$ now contains nodes that allow the agent to reach all other nodes in the graph.

With this problem out of the way, we thought of a way that wouldn't imply creating a Master Node. This led us to the idea of creating a classifier that would predict which nodes in $\mathcal{R}$ are the most likely to have access to **all** the key nodes. This had the advantage of reducing the complexity of the problem, as the agent would only have to choose from a subset of nodes, rather than all the nodes in the graph.

This time for the classifier, we used the Sickit-learn implementation of the Random Forest Classifier [sklearn]. We would input all the nodes in $\mathcal{R}$ to the classifier, and then let it predict which node would have a path to all the key nodes. We can then take the node with the highest probability as the root node. Also, another idea would be to sort the nodes by their probability and then let the agent start from the first one and then decide if it should go to the next one or not.

The features used for the classifier are :

- nb_nodes : The number of nodes in the graph
- $node\_degree$: The degree of the node to be classified
- $density$: The density of the graph
- $struct\_size$: Size of the corresponding malloc block
- $valid\_pointer\_count$: The number of valid pointers within the block
- $invalid\_pointer\_count$: The number of invalid pointers within the block
- $first\_pointer\_offset$: The offset of the first pointer within the block
- $last\_pointer\_offset$: The offset of the last pointer within the block
- $first\_valid\_pointer\_offset$: The offset of the first found valid pointer within the block
- $last\_valid\_pointer\_offset$: The offset of the last found valid pointer within the block

Once the classifier is trained, we save it to a file and then use it to predict the root node for the agent.

4.8 Reinforcement Learning

Now that we have set some tools for us to ease the conception of the agent, let's dive into the core of our project. In this section, we will formalize our State Spaces, Action Spaces, our rewards strategy as well as the design of our agent.

4.8.1 Environment Definition

Let us begin by defining our environment, and explain the motivation behind our choices.

Thanks to our Root Node Classifier, we can define from which node the agent will start exploring the graph. Thus, we only need to define the environment as a subgraph of the original graph, centered around the root node. This subgraph will be the environment in which the agent will navigate, and try to find the keys.

We create this subgraph by performing a Breadth-First Search (BFS) from the root node, thus creating a BFS tree. This has multiple advantages :

- Keeping only the nodes that are reachable from the root node.
- Removing any cycles or loops that might exist in the original graph.
- Assure that we always take the shortest path to the keys.

From now on we will refer to this subgraph as "The Graph".

Furthermore, one important assumption is that the keys will always be represented as leaves in the generated graphs, as the block containing the keys should only contain the key value without any other pointers. We say that the agent managed to *find the key* if the current leaf it is on is the corresponding key node.

4.8.2 State Definition

In this work, we have tried 3 state definitions. Each time we tried to increase the complexity of the state by adding information that we thought could help the agent to better find the keys.

1. Current Node Only: The state is only the features of the node where the agent is currently on.

2. Graph and Current Node: The state is the graph and the ID of the Current Node in this graph.
3. Graph, Current Node and Visited Subgraph: The state is the graph, the ID of the Current Node in this graph, and the subgraph that corresponds to the agent visited path.

Let us now explain the motivation behind choosing those states. First, the Current Node features seem mandatory, as they describe in some way the position of the agent in the graph. We thought of including the graph, as it could help the agent to understand the overall structure of the graph, and therefore learn patterns that could help it locate the keys. Finally, including the visited subgraph was motivated by the fact that the agent doesn't see the whole graph, even with graph convolution layers, getting the embedding of a node only takes into account a limited depth from the node, depending on the number of layers stacked. Therefore we got inspired by some works that dealt with Partially Observable Markov Decision Processes (POMDPs) [Zhu+17], where the agent doesn't have access to the full state of the environment and has to infer it from the observations it makes. This was also motivated by the thought that the more the agent explores the graph, the more it would be able to pinpoint where the keys are. Also, note that we could also input the visited nodes as a sequence rather than a subgraph. Though we didn't try this method.

4.8.3 Action Space Definition

In this section, we define our Action Space, and we also see why it wasn't as straightforward as we initially thought.

Intuitively, we want the agent to be able to move from node to node through their edges. Thus, the action consists of picking one neighbor of the node where the agent is currently on.

The primary challenge arises from the high variability in the number of a node's neighbors, which can sometimes be quite large. This variability complicates the use of neural networks, which require tensors with fixed attribute sizes.

In this work, we tried two different action spaces. We first began by implementing the agent with a Fixed Action Space, as it is the most common choice in Reinforcement

Learning. But this method while being straightforward comes with a few drawbacks in our scenario.

Fixed Action Space

As implied by the name, the Fixed Action Space needs us to fix the size of the action space.

One way of handling this problem is fixing the size to some sort of maximum K that would ideally fit every possible number of neighbors and masking the 'invalid' actions (Neighbors that don't exist). This method works but comes with a few drawbacks, the main one is that we have to somehow find the maximum number of neighbors, which forcefully impacts the capacity of our agent to generalize to graphs that could have some nodes that have an out-degree larger than this arbitrary maximum. Another limitation arises when the out-degree of the majority of the nodes is mostly quite small but with some "outliers" with very large degrees. For instance, if the majority of the nodes have a mean degree of 2, but still have some nodes with a degree >300. This forces us to set the Action Space Size to this maximum, this becomes limiting especially when using Neural Networks, due to an unnecessarily large overall complexity. To sum up, our Fixed Action Space is defined as:

$$A = \{a_i \mid a_i \in N \text{ with } 0 < i < K\}$$

Where N is the set of neighbors of the Current Node, a_i is the i^{th} neighbor of the Current Node, and K the Fixed Action Space Size (number of neighbors to choose from).

Furthermore, we could think that simply setting K to the number of neighbors of the current node might suffice. But it is possible that another node (from the same graph or not) has a higher out-degree, and thus the agent would not be able to generalize to this new graph.

Another drawback would be the difficulty for the model to associate which neighbor index corresponds to which output action. In our tests, we always tried to sort the edges by their $offset$ feature, which could help the agent learn the order of the neighbors. Though intuitively, this could impact the generalization capabilities, as it would choose the neighbors based on their $offset$ feature, rather than their actual features.

For this reason, we came up with the idea of using a Variable Action Space, which would allow us to overcome this limitation.

Variable Action Space

In this section, we explore the concept of a variable action space, which offers a dynamic range of actions specific to each state the agent encounters, enhancing the generalization capabilities of our agent.

A variable action space will allow us to choose from a set of actions that are not fixed. This means that the Action Space Size will resize at each State that the agent is currently in. Thus granting the agent the ability to generalize to any graphs, while avoiding the use of unnecessary memory space and Neural Network Complexity.

However, from this generalization capacity comes the price of larger training complexity. This method typically involves computing a value for each state/action pair. For example, if we are on a node (State) S, and this node has 3 Neighbors. We would have to compute a value for those 3 neighbors independently. While the Fixed Action Space could compute those values in one pass.

We decided to go with this method, as on top of allowing the agent to navigate through any graph, it predicts the Q-Values for each action independently, thus allowing the agent to choose the best action for the current state, without overfitting over the edge's $offset$ feature.

4.9 Goal integration

One of the main difficulties of our problem lay in the fact that we have multiple keys (target nodes) to reach. Thus the question of how to handle multiple of those naturally arises. While this might not seem like a huge problem at first sight, this proved to be more difficult to handle than expected.

4.9.1 Non Goal Oriented

Our first intuition was to create an agent capable of finding all the keys during only one episode. As we explained before, we consider that we reached a key, if the agent reached a leaf of the graph. While simple, this method comes with the limitation of not being able to detect exactly which key has been reached. To prevent that, we first tried fixing the order in which the keys should be reached. This involves:

- Changing the reward strategy, giving a positive reward only if it manages to find each key in the correct order.
- Modifying the state to include information about which keys have been found. In our case, we used a simple one-hot encoding of the keys that have been found. This allows the model to get a signal of which keys have been found and should help with generalization.
- Resetting the agent position to the initial root node, once it has found one key. Since our graph is a directed tree, and that our agent doesn't have any actions to go back to a previous node, we have to reset the agent to the root node, to allow it to explore the other branches of the graph.

Regarding the *resetting* of the agent position, we thought of another way, which would imply adding a "backtrack" action, that would allow the agent to go back to the previously visited node.

Adding a "backtrack" action could work however, since we have decided to use a variable action space, we thought that we needed to add some hard-to-implement complexity to the model by adding another output layer that could output discrete actions. Nevertheless, at the time of writing this, another idea came to mind, that wouldn't require adding another output layer. We would just need to *unmask* the previous node, and then let the agent choose it as the next node to visit.

Also note that the only reason that we can one-hot encode the keys in the state, is because we fixed the order in which the keys should be found.

To sum up, the main advantage of such a method is the ability to remember which keys have been found and thus could help in finding the other ones. However, this comes with the huge drawback of being extremely hard to train, as reaching one target key

node is already quite hard, and the agent has to reach all of them in the correct order. See chapter 5 for more details.

This main drawback is the reason why we decided to switch to a Goal Oriented method.

4.9.2 Goal Oriented

Let us now discuss the Goal-oriented method and the challenges that come with it.

This goal-oriented method comes from the wish to simplify the problem's complexity by allowing the agent to focus on one key (target node) at a time. This key node will be considered as the ultimate **goal** of the agent. This method works by adding the goal as an additional input to the agent model. We get the goals from the Graph Key Classifier, which predicts the number of keys in the graph, and then the keys that are present in the graph. To encode the goal, we used a classic one-hot encoding of the key. For example, for Key A, the goal encoding would be $[1, 0, 0, 0, 0, 0]$. This encoding is then stored alongside the state of the environment and passed as input to the model.

Integrating the goal method comes with various advantages. It allows the agent to train on a simpler problem, as it doesn't need to learn how to reach one target node before being able to reach the next one. It also allows the agent to learn how to reach each key independently, which not only should help with generalization, but also let us choose which key we want to find at production time. On top of also removing the hassle of choosing how to reset the agent position, we can easily retrain the agent on a particular key if needed.

However, this method also comes with a few drawbacks, the obvious one being that making the exploration process independent from one key to another, means that it cannot utilize its exploration knowledge of the previous keys to find the next ones.

We then have to revisit the training strategy. We now need a way to train the agent on multiple goals, while making sure that it isn't biased towards any of the goals. There are multiple ways to achieve this, we could imbricate the main training loop in another loop, that would iterate over all possible keys. This way we would make sure that the agent has the same amount of training on each key. Another way would be to uniformly sample the goal at each episode, this way we would make sure that for a large enough number of episodes, the agent would have **sampled** all the keys the same amount of

time. At first, we decided to go with the latter, as we hoped that it would remove any "ordering" bias that the previous method could had, and thus help with generalization.

However, one important thing to note is that sampling uniformly, and therefore sampling the same amount of time for each key doesn't necessarily mean that the agent will experience reaching each key in the same amount of time. For instance, we often observed a behavior where the agent would manage to reach a particular key quite a few times early in the training, making him more confident in reaching this particular key the more he trained. Therefore in the experience buffer, we would have more experiences of reaching this key, and thus the model would be more biased towards this key. This would make the agent try to go for this particular key, even if the goal is another one.

To try and prevent this behavior, we first tried to change the probability law of the goal sampling. We created a custom probability law, that would give more chance to the keys that have been reached the least amount of time. This method helped when we trained for a relatively large number of episodes in one environment.

However, if we limited the number of episodes to a couple hundred per environment, the same problem as the previous method would arise.

We then decided to switch strategy, and always choose the least reached key as the goal. This seemed to perform better in the case where we limit the number of episodes per environment.

4.10 Variable Action Space Deep Q Model

In this section, we discuss our main method, which uses a Variable Action Space and Graph Neural Networks.

4.10.1 Model Architecture

Let us now discuss the design of our model architecture.

As explained previously, we allow the agent to choose from a variable number of actions, by computing a Q-Value "one by one" for each possible action (Node in our case). So the output would be a scalar Q-Value for each possible action, and we would then choose

the action with the highest Q-Value. As we use Graph Neural Networks and PyTorch Geometric data structure, we compute a Q-Value for each node in the graph, and then we mask the invalid actions by setting their Q-Value to $-\infty$.

To better explain our model architecture, let us first define the input of the model.

The inputs of the model are the following :

- x : The node features of the graph
- edge_index : The edge index of the graph, defines the connections between the nodes
- edge_attr : The edge features of the graph ($offset$)
- current_node : The ID of the current node
- visited_subgraph : The indices of the nodes in the subgraph, that corresponds to the agent visited path
- goal : The one-hot encoding of the goal

Let us now define the high-level view of our model architecture. Our model architecture mainly relies on the GATv2Conv [Gv2Conv] layer from the PyTorch-Geometric library, this choice is motivated by the fact that it calculates an attention score for each neighbor of a node. As such we hope that it would help the agent focus on the most important neighbors to reach the goal. We first apply some GATv2Conv layers [Gv2Conv] to the graph, to get the node embeddings. let us denote this output as h.

Then we get a global embedding of the graph, by applying a global mean pooling layer [GMP] to the node embeddings, let us denote this as g.

We then also get a global embedding of the subgraph, by getting all individual node embeddings in h that correspond to the visited subgraph, and then applying a global mean pooling layer to those embeddings, let us denote this as s.

We then get the embedding of the current node, by selecting the corresponding node in h, let us denote this as c.

Finally, we concatenate all those embeddings, let us denote this as z.

Now, we need to integrate the goal into the model. For this, we created some goal-specific dense layers, which means that depending on the goal, z will go through different dense layers. The motivation behind this rather than simply concatenating the goal with z, is

that we want the model to better differentiate the path to take to reach the goal. The output of this layer will be our predicted Q-Values for each possible action.

For the sake of simplicity, we omitted some details, such as the activation functions, batch handling, normalization, and masking.

4.10.2 Agent

In this section, we discuss the design of our agent and the methods we used to train it.

Reward Strategy

Along the development, we have tried numerous reward strategies, each of them depending on what action we allow the agent to take. In our latest implementation, we decided to go with a relatively simple reward strategy.

- reward DISTANCE_REWARD for getting closer to the current target (goal) node.
- reward KEY_REWARD for reaching the target node.
- penalty NO_PATH_PENALTY for taking an action that leads to a node that has no path to the target node.
- penalty WRONG_KEY_PENALTY for reaching a leaf that is not the target node.

Note the fact that we don't introduce any penalties for getting farther away from the target node. This is because the graph has been created using a Breadth-First Search, thus the agent isn't able to "go up" in the tree.

The choice of the values of those rewards and penalties can have a great impact on the training process. Too extreme values could lead to the gradients exploding. Too low values could lead to the agent not learning anything. For our case, we decided to go with the following values :

- DISTANCE_REWARD = 4
- KEY_REWARD = 10
- NO_PATH_PENALTY = -10

- WRONG_KEY_PENALTY = -10

Experience Replay Buffer

The use of a replay buffer is quite common in Deep Reinforcement Learning. In the simplest form of training for Deep Q-learning, we only perform a training step after each action taken by the agent. This process has numerous disadvantages, on top of being slow and not taking advantage of the batching capabilities of modern GPUs, the experiences become highly correlated. Leading to a high variance in the training process. Therefore making the model harder to converge. The classical experience replay buffer allows us to store the experiences of the agent, and sample them uniformly to train the model. This method has the advantage of breaking the correlation between the experiences, hence a more stable training. This approach also enables the model to revisit previous experiences multiple times through batching, which can significantly accelerate the training process.

For the experience replay buffer and sampling, the most common and simple way is to use a simple fixed-size array that we fill with every experience, and that we sample uniformly from during training. This method has the advantage of being fast to implement and fast to execute.

However, for our problem, we decided to go with a **Prioritized Experience Replay(PER)** [Sch+15] Buffer instead. The main idea behind this method is that rather than saùpling each experience with the same probability, PER uses the **td-error** (difference between the predicted Q-Value and the target Q-Value) to compute a prioritization score. This prioritization score will be derived to have a sampling probability, giving more chance to experiences that had larger td-errors to be sampled during training.

Now, the main motivation for choosing this method rather than the previous one lies in the fact that in our case, the rewards are extremely sparse. This sparsity can lead to the majority of the experiences having the same low signal reward. This has the inconvenience of giving less chance to the high-reward experiences to be sampled, which will, in turn, bias the model into predicting this common low reward most of the time. Thanks to PER, we give more chances to experiences with high rewards to be sampled, as their td-error will likely be greater than the penalty ones. Note that those priorities are updated during training, meaning that as the loss of some experiences decreases, their

priority, and as such their sampling probability also decreases. This method creates a bias towards those high-loss experiences, thus the original paper advises using **Importance Sampling(IS)** to correct this bias over time. However we decided to omit the effect of this bias, as finetuning the hyperparameters was quite hard, and that it overall didn't seem to help much. In terms of pure implementation, we went with a Sum Tree data structure [SumTree] as it allows us to sample experiences with a complexity of $O(\log(n))$, where n is the size of the buffer.

Exploration Strategies

To avoid being stuck in a local optimum, we decided to use a regular ϵ-greedy exploration process. Where with probability ϵ we will either choose a random action (Exploration) or select the action with the higher predicted Q-Value (Exploitation). We anneal ϵ by a certain factor throughout the training process. Privileging exploration at first and aiming toward exploitation near the end. This choice is motivated by the fact that it is the most common exploration strategy for Deep-Q-Learning, but also by the sparse nature of our problem. There are usually more paths unable to reach the key nodes, than the inverse. Thus limiting the amount of reward signals. A stochastic exploration process tends to converge too fast and thus doesn't allow for efficient exploration of our environment. The ϵ-greedy exploration process allows us to manipulate how much the agent should perform random walks rather than exploiting the predicted Q-values. Though fine-tuning the annealing of ϵ can be quite hard, and needs to be done with care.

Training

In this section, we discuss the training process of our agent.

We use a *Double* Deep Q Network method [HGS16]. This method improves over Deep Q Networks by addressing the overestimation of the Q-Values, in order to stabilize the learning process. In standard DQN, the same network is used both to select the best action and to evaluate its value, which can lead to over-optimistic value estimates. Double DQN addresses this issue by decoupling the selection of actions from their evaluation. This implies the use of **two** networks, one for action selection (policy network) and one for value evaluation (target network). The policy network is updated at each step, while

the target network is updated less frequently. Thus, the target network won't be updated through backpropagation. There are usually 2 methods used to update it, either through what we call a hard update, which means that for every τ step, we will copy the weights of our DQNN to the Target one, or either through a soft update which updates the weights of the target after each loop iteration, but with a linear combination of the weights of the two models.

Also, as explained in chapter 2, the Deep-Q-Neural-Network goal is to predict the expected reward, this value is called the Q-Value, and we use the predicted Q-Value as input to our policy. Our policy is deterministic, as it will always choose the action that has the highest predicted Q-value. Thus once our model is trained, for the same inputs, our agent will always perform the same actions. This means that our model can't gain knowledge through multiple executions, if it fails once, it will always fail on this input.

Furthermore, one important aspect that we had to think through was the way to handle batching. Batching is quite important, as it allows us to fully utilize the capacity of our CUDA device to speed things up, avoiding the latency from transferring data from the CPU to the GPU memory. However, batching with Graph Neural Networks is not as straightforward as other Neural Networks. Usually, in a classic Neural Network, we need the input to always have the same shape within a batch. The main difficulty here lies in the fact that the size and shape of the graphs can vary from one to another. Therefore we needed a way to handle this problem.

PyTorch Geomtric already provides a way to handle this batching with graphs, by using a Graph Data Loader. This data loader will take a list of PyTorch Geometric Data objects and will batch them in a way that is compatible with the Graph Neural Networks.

The way it handles this is by creating one large graph that concatenates all the graphs in the batch along the node dimension and then provides us a mask that will tell us which nodes belong to which graph. Thus, we have to take this mask into account when designing our model. One interesting fact is that the Graph Convolution Layers don't need the mask, thanks to the message passing mechanism, it will only aggregate the features of the neighbors of each node and thus will ignore the features of the nodes that don't belong to the current graph.

However, we encountered a significant limitation related to the time required to transfer data batches to the GPU. This time can be substantial, especially when dealing with large batch sizes and relatively large graphs. Although it may take 'only' one second to

perform one training iteration, this time accumulates rapidly over many episodes. One potential solution is to improve how graphs are stored for each experience. In our current implementation, we store a copy of the same graph for each experience in one training file, with each copy being loaded into the GPU memory before training. This approach uses an unnecessarily large amount of memory. Ideally, we should store the graph only once per file, load it into the GPU memory, and then reference it for each experience. This method would significantly reduce both memory usage and the time required to transfer data to the GPU, thus enabling larger batch sizes and faster training. However, implementing this solution would require substantial refactoring of our code, and we leave it for future work.

5 Challenges Encountered

Throughout the development of this project, we encountered numerous challenges and difficulties. This chapter will detail the specific problems we faced and discuss the solutions we implemented to overcome them.

5.1 Graph Generation

The first challenge we encountered was about the Pointer Graph Generation program. Initially, the program was developed as a script written in Python. The simplicity of Python facilitated the rapid prototyping of various generation algorithms. Despite this, the performance of the final algorithm was underwhelming. Particularly, the time required to construct the graphs was excessively long. As such we decided to rewrite the program in Rust, which is a programming language that is known for its performance and safety. This decision was also motivated by the opportunity to tackle a significant technical challenge and to work with a new technology. On top of that, the first Python version had an issue that was not easy to fix without a complete rewrite of the program. The advantages of using Rust are numerous, primarily its compiled nature and support for multi-threading, which enables us to fully utilize the multiple cores of our processor. Running the graph generation with the compiled program of the Rust code performed very well, and largely outperformed the Python version.

5.2 Graph Neural Network Over-smoothing

Graph Neural Networks are often prone to what we call over-smoothing [Wu+23; Over-smoothing]. As explained in the previous sections, Graph Neural Networks are comprised of Graph Convolution Layers. A Graph Convolution Layer works by aggregating the

features of the neighbors of a node to update its features. The problem arises when stacking multiple of these layers. As the number of layers increases, the features of the nodes become more and more similar. This comportment can lead to over-smoothing, which means that it becomes harder to distinguish nodes based on their features. In our case, this phenomenon was particularly problematic and hard to debug, as there are no errors per se, the model is just doing what it is supposed to do.

Identifying this problem was challenging, but a small hint emerged from observing our agent's behavior. Even after quite a few episodes, the agent always seemed to behave the same, always choosing the same nodes and paths to explore. It is when we decided that rather than studying the agent's behavior, we would examine the actual predicted Q-values of the model. This showed us that the predicted Q-values of the direct neighborhood of the agent's current node had pretty much the same values.

At the time, we didn't know about over-smoothing, as such we came up with some hypothesis that would make the model behave this way. Namely, we tried to remove the dropout layers, as we thought it could have been a generalization issue. Even though it seemed to help a bit, the issue was still present. We then thought that there was a problem with the features themselves, so we changed our state representation, by trying to give it more information, like the degrees of the nodes, some centrality score, and more generally more graph-specific features. This also didn't help, and we were still stuck with the same issue.

It is only after reading some articles on the internet about Graph Neural Networks, that we came across the concept of over-smoothing. This was a revelation, as it perfectly described the issue we were facing. It seemed to be a common problem, and some solutions were proposed. The ones that we tried out were :

- Adding skip connections between layers
- Increasing the convolution layer's output dimension
- Graph Normalization

Other solutions exist, including some intriguing methods that minimize this effect by directly affecting the loss function; however, we chose not to pursue these.

To sum up, adding all of those solutions helped a lot, the model managed to efficiently differentiate the nodes, and as such the agent performed much better.

5.3 Exploration Strategy and Sparse Rewards

A common and very important aspect of Reinforcement Learning is the exploration strategy. It defines how the agent will act in the environment to maximize the number of relevant observations. As mentioned before, the goal here is to balance between exploration(trying new things) and exploitation(using what we already know). This is an especially important aspect of the problem we are trying to solve, as the environment is not deterministic. Of course, the version of OpenSSH as well as the time at which the Heap Memory has been dumped, can affect the size and shape of the graph generated from this Heap Dump File.

Furthermore, our problem is also a sparse reward problem. If we consider that we only give a reward when the agent finds the target nodes, then the agent will have to explore a lot before finding one target node, thus the replay buffer will mostly contain experience with basically no rewards. This is problematic as the agent model will tend to overfit on the no reward experience, and as such will not be able to generalize well on the reward experience. Even if we use a distance-based reward, to try to augment the number of rewards, those remain rather sparse, as often, there aren't many paths leading to the target nodes. Thus, most of the time, the agent will explore paths that do not lead to the target nodes, and will then overload the replay buffer with "no reward experience" in the same way.

To tackle this issue, we tried to implement different strategies. The first one was to use a distance-based reward, as explained before. This helped a bit, but the rewards were still quite sparse. Then, we implemented the famous Prioritized Experience Replay, allowing us to have more chances to sample experiences with rewards during training. And finally, we noticed that the main issue was that the generated graph was indeed very large, this is why we created the "ideal" root prediction model, as described in the previous chapter. The goal of this model is to reduce the size of the problem, by only using a subgraph that we hope, contains paths to all the target nodes.

Speaking of training, at first we were under the naïve assumption that the main reason why the agent was not learning, was **only** caused by the complexity and large exploration space of the problem. Thus, when we first started creating the training loop, we considered limiting the number of actions the agent could take to speed up the training. As such we implemented an early killing mechanism. It would terminate the episode if

the agent chose an action leading to a node with no paths to any key nodes. At first, we were satisfied with this method, because it helped with the training speed. However, it did not help the model to learn better. Thus we had the intuition to try and remove this mechanism as we thought it could limit the amount of data diversity that the agent's model could see. And indeed, after removing this mechanism, the agent started to learn much better. This was a great lesson for us, as it showed us that sometimes, trying to make a model learn faster, doesn't necessarily means it will make it learn better.

On top of that, we also applied a BFS tree from the predicted root. Thus it would ensure that the agent would only choose the shortest path to the target nodes, as well as remove any loops or cycles. We left this in the final version of the agent, though we reckon that it could be removed. The main reason for thinking this is that similarly to before, it could limit the amount of diversity in transitions that the agent could see (because it removes quite some edges, each node only has one parent). And also, it is semantically wrong. Each node corresponds to a Struct in the Heap Memory, and each node contains features about its size, and the number of pointers in it. As such, a node may have multiple parents, but the BFS tree would only allow one. Thus removing some edges that are specific to the Heap Memory, implies altering the semantics of the graph. This could have an impact on the agent's learning. However, this hypothesis is yet to be tested.

Furthermore, as explained before, we decided to use the classical ϵ-greedy strategy. But for the training of the Agent, we had to decide on how to apply this strategy. The main reason this decision was not straightforward was because we trained the agent on multiple graphs (Environments). As such we had two options :

- Apply the ϵ-greedy strategy on each graph independently
- Apply the ϵ-greedy strategy on the whole batch of graphs

Both of them come with their advantages and disadvantages. Applying the strategy on each graph independently (resetting epsilon after each environment) would allow the agent to explore more in a more "efficient" manner, as explained before the rewards are sparse, as it implies going pretty deep in the graph to find the target nodes. This method could allow it to first perform a random walk, learn from it, and then give it the ability to exploit this and have the chance to reach the target nodes as epsilon decreases. One of the main problems that we noticed using this method, was that, interestingly, it influenced the model gradients in a pretty bad way. Even though we didn't manage

to exactly pinpoint what caused this issue, we intuitively think that the agent had time to "converge" to a global minimum for this particular environment, and then when the environment changed, the model was not able to generalize to this new environment, thus causing large gradients and a bad learning process.

Applying the strategy to the whole batch of graphs would allow the agent to learn more gradually, as it will mainly be doing random walks in the first few graphs, and then exploit this knowledge in the later graphs. This method proved to be more stable, and we managed to obtain quite decent results with it. However, we had to make sure to properly finetune the decay rate and shuffle the graphs in the batch. The main point that worried us was the fact that the first few graphs wouldn't be properly explored, as it is very unlikely that it got a lot of rewards for this particular environment, and that during the testing phase, it wouldn't perform well on those. That this why we hoped for the generalization capability of the model.

To illustrate the problem of sparse rewards, let's assume a relatively small graph in a tree form, with each node having 5 neighbors and a depth of 4. Taking into account that the target node is a leaf, we know that we have to go through 4 nodes to reach the target node from the root. To illustrate the problem, with a pure random walk, the probability of reaching this target node is $(1/5)^4 = 0.0016$. This in turn means that for 1000 episodes, the agent would only reach the target node 1.6 times on average.

Also, we have to take into account that the early decision of the agent has a huge impact on whether it will reach the target node or not. If we assume that the agent perfectly learned which node to choose at each step (which is not the case at the beginning of the training), then the probability of selecting the correct node (the node that gets the agent closer to the target node) is $1 - \epsilon$. We can then assume this problem as the Bernoulli trial :

- The success S is the event that the agent chooses a node that gets it closer to the target node
- The failure $\overline{S}$ is the event that the agent chooses a node that doesn't get closer to the target node

with $P(S) = 1 - \epsilon$ and $P(\overline{S}) = \epsilon$.

With our previous example, we know that we have to perform 4 trials to reach the target node. Thus the probability of reaching the target node is $P(S)^4 = (1 - \epsilon)^4$.

This further highlights the fact that overall, the number of experiences that can manage to reach the target node is very low, which justifies the sparse reward problem. This problem becomes even more problematic as the graph size and depth of the tree increase.

At one point in development, we also had an idea to try and mitigate the sparse rewards problem. We thought of using a technique called Hindsight Experience Replay (HER) [And+17]. Which is essentially a technic that allows the agent to learn from experience that didn't lead to a reward, by pretending that it was the goal of the agent all along. For example, we can think of a robot trying to reach a goal, the goal being a specific point in the environment space. If the robot reaches the goal, then it gets a reward, if it doesn't, then it doesn't get any reward. In this case, the action space is continuous, the policy has to learn the angles and the speed at which the robot has to move to reach the goal. So the problem here is very sparse, as the number of experiences where the robot reaches the goal purely from random exploration is very low. Thus, HER allows the agent to transform "failed" experiences into successful ones, by switching the goal of the agent to the state where the robot reached. Accelerating the learning process and enhancing the agent generalization capabilities.

In our case, we thought of applying this very same technique to our problem. We could pass the target node (Goal) as a parameter to the agent model, and then, when it reaches any node that is not the target key node, we could switch the initial goal by this very node. This would help with the sparse reward problem and would allow the agent to learn more efficiently. However, some problems arose when we tried to implement this technic.

First of all, we had to define a way to represent the goal. In the case of a robot arm, for example, the goal would be a point in 3D space, represented as X, Y, and Z coordinates. But in our case, this implies that we know what the target is. While we might know for which key we are searching (A, B, C, D), we don't know to which node the key corresponds, it is the very problem we are trying to solve with our method. So at first, we thought of encoding the target node as a one-hot vector, for each of the 4 possible keys. But then we realized that it wouldn't work, as the agent may reach a leaf that is not a target key node. Another idea that we had was to create some sort of model that would learn the state feature representation of each target key node, and thus when we reach a node that is not a target node, we could replace the goal with the features of the current node that we reached. While this idea seemed interesting, creating a model

that would do some sort of regression to predict the target node features seemed a bit overwhelming, thus we decided not to go further with this idea.

On the other hand, we still tried to implement some variants of the HER technic. Since our problem has multiple goals, if our agent reached a target node, that was not the initial goal, we could this time, switch the goal to this other target node. While this technique doesn't work for all the nodes, it would still at least increase a bit the amount of overall successful experiences and thus would help the agent to learn more efficiently. At least, this is what we thought. While it indeed increased the number of successful experiences in the replay buffer, it came with one huge drawback. The agent wouldn't have any experiences where it failed to reach its target goal node, thus it would always give a high predicted Q-value to any target node, even if it is not for the specified goal. We tried to mitigate this by adding some sort of probability of switching the goal, but in the end, we decided to completely remove this technique, as it contributed to some overfitting of the model.

5.4 Problem Definition

Before actually implementing the project, we, of course, had to properly define the problem we were trying to solve and the way we were going to solve it.

At first, we thought of simply creating one agent that would be able to :

- Efficiently explore the graph
- Detect which nodes are the key nodes
- Stop when all the key nodes have been found

We hoped that our agent would be able to learn all of this by itself, but this also came with some challenges.

When we started to implement the agent, we decided to go step by step. So in the first implementation we made, the agent didn't have any key node detection mechanism or any stopping mechanism. Those were hardcoded in the environment itself, as that information was available in our training dataset. However, this wouldn't be possible in a real-world scenario, as that information is **only** available in the training dataset. The

idea here was to first make sure that the agent was able to explore the graph properly, and then ideally expand it to perform the key node detection and stopping mechanisms.

As explained just before, we wanted to first make sure the agent would be able to learn how to explore. So in the first iterations of the development, we focused on just that. We took a simple subgraph, that we know contains paths to all the target nodes, and we trained the agent on this subgraph. We first gave the agent rewards if it reached any target nodes, independently of the order in which it reached them. And we hard-coded the stopping mechanism, by stopping the episode when all the target nodes have been found. This allowed us to make sure that the agent was able to learn how to explore the graph properly, and that it was able to learn how to reach the key nodes.

This is when we first had to decide on how to "detect" the target nodes. Essentially, our problem is a mix of an exploration problem and a "classification" problem. We want the agent to be able to explore the graph to reach the key nodes, but also be able to tell if the node it reached is a key node or not. So we had a few ideas in mind to tackle this problem:

- Add some sort of special action to the agent, that would the policy would trigger once it thinks it has found a target node.
- Add a classifier to the agent, that would take the state as input and output a probability of this state being a target node.
- Using Hierarchical Reinforcement Learning, where the agent would have two policies, one for the exploration and one for the target node detection.

While all of those solutions could work, we made a decision based on a simple observation. Which is that all the target nodes, must be leaves of the graph. Thus if we manage to teach the agent to efficiently explore the graph, we could assume that if the agent reaches a leaf, then it must be a target node. This assumption made the whole problem much simpler, as we didn't have to make the agent model even more complex.

The main drawback of this method is that we have to make sure that the agent is extremely reliable, as it doesn't give any wiggle room for errors.

5.5 Reinforcement Learning Algorithms

For the agent to learn how to explore the graph, we had to choose a reinforcement learning algorithm. So we mainly tried two of the most famous ones, Soft Actor-Critic [Haa+18] and Deep Q-Learning. At first, we tried to implement the Soft Actor-Critic (SAC) algorithm, as it has been shown to perform well on continuous and large action space problems. But the more we tried working with it, the more we realized that the sparsity of the rewards was a big issue for it. Since the main exploration strategy of the SAC algorithm is stochastic, the policy would often converge too quickly to a suboptimal policy, and thus wouldn't explore enough. We tried to tweak some hyperparameters, like the learning rate or even the entropy coefficient, but it didn't help much. Most of the time, while lowering the learning rate seemed to help at first, it would come a time when the agent would stop learning, forget what it learned, and perform even worse than before. We tried this method with both the Graph Neural Networks and without, but the results were not satisfying. This, of course, doesn't mean that the SAC algorithm is not suited for this problem, but rather that we didn't manage to make it work properly. We also tried the REINFORCE algorithm, which contrary to Deep Q-Learning, doesn't predict the cumulative expected reward, rather it directly tries to predict the actions that would maximize the expected reward. It directly maps states to actions, the core idea is to maximize the expected reward by directly optimizing the policy. We say that the REINFORCE algorithm is policy-based, while the Deep Q-Learning algorithm is value-based. The same observation and problematics of the SAC algorithm arose with the REINFORCE algorithm. We are convinced that with more time and finetuning, those algorithms could perform well on this problem.

Those are the reasons why in our final implementations we decided to go with the Deep Q-learning algorithm.

In terms of implementation, we wanted to try some of the famous Deep Reinforcement Learning libraries, namely Stable Baselines. However, we quickly realized that it would be hard to implement our own Graph Neural Network model with it, as it is not natively supported. Stable Baselines also assume the environment to be fixed-sized, which is not the case for our problem. So we decided to implement all of our algorithms and models from scratch, using PyTorch.

To conclude, throughout the development of this project, we encountered numerous

challenges, both technical and theoretical. Addressing these issues provided a valuable learning experience. The most frustrating aspect, however, was the interdependence of every project component. Often, solving one problem would inadvertently create another, leading to a continual cycle of adjustments across different components. This resulted in a significant time spent observing the agent during training, only to discover it had not learned as expected, prompting us to investigate and understand the reasons behind these outcomes.

6 Results and Discussion

In this chapter, we present the results of our experiments and discuss the performance of our models.

6.1 Root Predictor

In this section, we present the outcomes of our Root Predictor analysis. We utilized a dataset comprising 12,523 graphs to assess the results generated by our Root Predictor.

In Table 6.1 the performance of the Root Predictor model is evaluated using a range of metrics including accuracy, precision, recall, and F1 score. These metrics offer valuable insights into the model's capability to accurately identify probable roots, categorizing them either as "isolated from all the key nodes" (Class 0) or "Connected to all the key nodes" (Class 1).

The Root Predictor demonstrated a remarkable overall accuracy of **100%**, underscoring its high effectiveness in root classification throughout the dataset. The specific performance metrics are detailed as follows

Metric	Value
Accuracy	1.00
Recall (Macro Average)	1.00
Precision (Macro Average)	1.00
F1 Score (Macro Average)	1.00

Table 6.1: Summary of model performance metrics

Table 6.2 shows the classification report for the Root Predictor model, which provides a detailed breakdown of the model's performance across the two classes.

Class	Precision	Recall	F1-Score	Support
0	1.00	1.00	1.00	263015
1	1.00	1.00	1.00	7720

Table 6.2: | **Classification Report for the Root Predictor Model.** The "Support" column indicates the number of *possible root nodes* of each class in the test set. The results indicate that the model performed exceptionally well in classifying roots into the two categories. The high precision and recall values for both classes suggest that the model was able to accurately identify the root nodes that have paths to all the target key nodes in the majority of cases.

The confusion matrix in Table 6.3 further illustrates the model's classification behavior:

	Predicted Class	
Actual Class	Class 0	Class 1
Class 0	263015	0
Class 1	0	7720

Table 6.3: Detailed confusion matrix of the model

The confusion matrix shows that the model correctly classified all instances of both classes, resulting in a perfect score for accuracy, precision, recall, and F1 score.

The model managed to achieve a perfect score across all metrics, indicating that it was able to accurately classify the root nodes in the dataset.

However, it is important to note that this model was tested on a dataset that contained the same OpenSSH version as the one used for training. Thus, we don't have any information on how the model would perform on unseen OpenSSH versions.

We guess that the multiple generated graphs for one OpenSSH version are relatively similar, thus it is possible for the model to "overfit" on those versions. This is why it is important to test the model on unseen OpenSSH versions to ensure its generalization capabilities.

6.2 Key Count Prediction

In this section, we will now analyze the results of our Key Count Prediction model. The Key Count Prediction model was trained on a dataset of 8317 graphs, as a reminder,

the goal of this model is to predict between 3 classes :

- Class 0: Key Count = 2
- Class 1: Key Count = 4
- Class 2: Key Count = 6

This prediction also allows us to determine **which** keys are present in the graph.

For the test set, we used a different dataset comprised of 12523 graphs. This dataset comes from the "Validation" folder provided with the SmartKex work [Fel+22]

The Key Count Prediction model achieved a perfect score of **100%** overall metrics on the whole 12523 test graphs, as shown in Table 6.4. The detailed performance metrics are as follows:

Metric	Value
Accuracy	1.00
Recall (Macro Average)	1.00
Precision (Macro Average)	1.00
F1 Score (Macro Average)	1.00

Table 6.4: Summary of model performance metrics

Without surprise, the classification report in 6.5 for the Key Count Prediction model shows a perfect score for all classes.

Class	Precision	Recall	F1-Score	Support
0	1.00	1.00	1.00	2972
1	1.00	1.00	1.00	5816
2	1.00	1.00	1.00	3735

Table 6.5: | **Classification Report for the Key Count Predictor Model.** Report shows a perfect score for all classes.

As we can see from the results in Table 6.5, the Key Count Prediction model performed exceptionally well on the test dataset, achieving a perfect score across all metrics.

Similarly to the Root Predictor model, the Key Count Prediction model was tested on a dataset that contained the same OpenSSH version as the one used for training. Thus, we don't have any information on how the model would perform on unseen OpenSSH versions.

Intuitively, we would assume that this model would generalize better than the Root Predictor model, in part due to the use of Graph Neural Networks.

6.3 Variable action space Goal based Deep Q Learning

In this section, we will inspect the results of our Variable action space-based Deep Q Learning model. We decided to only show the results of this model as it is the one that we consider the most interesting and the most promising. This model allows us to go beyond the limitations of the fixed action space of a regular method, and to learn a general policy that could be used on any graph shape and size.

For the following section, when we refer to "the graph", we are referring to the subgraph of the globally generated graph which is the Breadth First Search (BFS) tree from the predicted root node. The model has been trained on 7 files of each of the subfolders present in our training dataset, for a total of 8317 files. We used a variable number of episodes for each file, depending on the number of nodes in the graph. More precisely, we used 1000 episodes for graphs with less than 200 nodes, 4000 episodes for graphs with less than 400 nodes, and 6000 episodes for graphs with more. This variable number of episodes has been motivated by the fact that each environment can be drastically different and that the model needs to explore more to learn a good policy.

We tested our method on the "Validation" dataset, which contains 12523 graphs. Furthermore, for this work, the model has been trained and tested to only predict the Initizialitation Vectors (Key_A and Key_B) as well as the Encryption Keys (Key_C and Key_D). As for decrypting a communication, the Integrity Keys (Key_E and Key_F) are not needed.

The following metrics are used to evaluate the capacity of the model to correctly find the goal key.

In the Figure 6.1, we can see an example of the real predicted Q-Values for a graph. This gives an idea of the reward function used by the model, and how it is able to predict the Q-Values for each action. One can see that the model can pretty accurately differentiate between the different actions and that it can predict the Q-Values for the goal key with a high value.

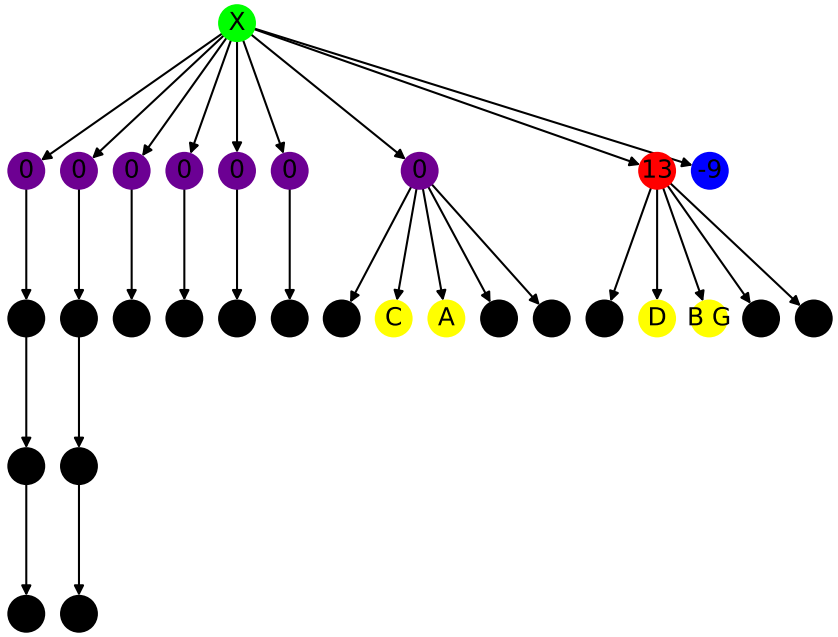


Figure 6.1: | **Example of the real predicted Q-Values for a graph.** The green node with an **X** represents the current position of the agent, here the agent is at the predicted root node. The neighbors are colored such that blue represents the lowest Q-Value, and red the highest. The yellow node represents the target nodes. Here "B G" represents the current goal key to be reached by the agent.

Goal	Precision	Recall	F1-Score	Support
A	0.98	0.95	0.97	9551
B	1.00	0.93	0.96	9551
C	0.99	0.95	0.97	12523
D	1.00	0.96	0.98	12523
micro avg	0.99	0.95	0.97	44148
macro avg	0.99	0.95	0.97	44148
weighted avg	0.99	0.95	0.97	44148

Table 6.6: Classification Report showing model performance across different goals

Our model achieved an overall accuracy of **95%** on the test dataset.

The classification report in 6.6 shows the performance of the model across different goals. The model achieved high precision, recall, and F1 scores for all goals, indicating that it was able to accurately predict the goal key in the majority of cases. However, we can still notice that the model has a slight bias towards the goal B and D.

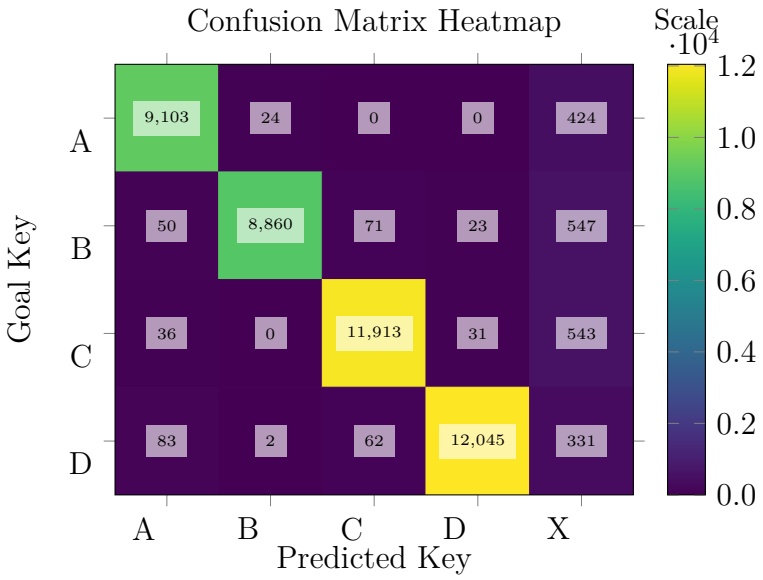


Figure 6.2: Heatmap of the confusion matrix showing the frequency of predictions across true goals

The heatmap in 6.2 provides a visual representation of the confusion matrix, showing the frequency of predictions across true goals. The "predicted goal" **X** represents the cases where the model reached a leaf node that is not a goal key. We can see that the

model still tends to predict incorrectly towards X. We can also notice a slightly better performance towards the goal C and D.

	Size 16	Size 24	Size 32	Size 64
basic_V_6_8_P1	1.00	1.00	1.00	0.92
basic_V_6_9_P1	1.00	1.00	1.00	1.00
basic_V_7_0_P1	1.00	0.68	1.00	0.87
basic_V_7_1_P1	0.99	0.69	1.00	0.87
basic_V_7_2_P1	0.98	0.74	0.84	0.70
basic_V_7_8_P1	1.00	0.78	1.00	1.00
basic_V_7_9_P1	1.00	1.00	1.00	1.00
basic_V_8_0_P1	1.00	0.00	1.00	1.00
basic_V_8_1_P1	1.00	0.97	1.00	0.99
basic_V_8_7_P1	1.00	1.00	1.00	1.00
basic_V_8_8_P1	0.95	0.73	0.97	1.00
client_V_7_8_P1	1.00	1.00	1.00	NA
client_V_8_0_P1	1.00	1.00	1.00	1.00
scp_V_7_8_P1	1.00	1.00	1.00	1.00
scp_V_8_0_P1	1.00	1.00	1.00	1.00

Table 6.7: | **Success Rates by Size and Process-Version.** The model achieved a high success rate for most process versions and key lengths, with a few exceptions where the success rate was lower. However, we can notice that our model had a success_rate of 0.00 for the basic_V_8_0_P1 with size 24. Apart from that, the process that had the most trouble was the basic_V_7_2_P1, which had a success rate of 0.70 for size 64, and overall lower success rates for the other sizes. client_V_7_8_P1 has a Non-Available (NA) success rate for size 64, as there was no data available for this process version.

Finally, the table in 6.7 shows the success rates of the model across different key lengths and process versions. The model performed well across all goals, achieving high precision, recall, and F1 scores for each goal key. The model was able to accurately predict the goal key in the majority of cases, with a slight advantage towards the goal C and D. One notable remark is the very low success_rate of 0.00 for the basic_V_8_0_P1 with size 24. After verification, this sucess_rate isn't perfectly equal to 0.00, but is very close to it. Also, the subgraph generated from the BFS tree of the root node still seems to have produced a rather complex graph to explore, with a long path to the target key nodes, and nodes with high degrees. This could suggest that there wasn't enough exploration done by the model to learn a good policy on this graph.

Moreover, we can see that the amount of data is a bit unbalanced towards the keys

C and D, which could explain the slight bias towards those keys. This unbalance is unsurprising, as every file will contain the keys C and D (Encryption Keys), but not necessarily the keys A and B (Integrity Keys).

Furthermore, the same reasoning from the previous models applies here, the model was tested on a dataset that contained the same OpenSSH version as the one used for training. Thus, we don't have any information on how the model would perform on unseen OpenSSH versions.

Nevertheless, we managed to obtain a model, that was trained on only 7 files per subfolder (All combinations of Process, Version, and Key Length) for a total of 8317 files, that can reach the goal key with a high success rate, which proves its capacity to generalize on unseen data.

It should be noted that the model could potentially be trained using fewer files, we let the reader experiment with the model to see if it is possible to achieve the same results with less data.

7 Conclusion

This thesis has advanced research in digital forensics by developing a method to extract SSH keys from heap dumps using graphs and deep reinforcement learning. Our innovative approach is designed to train on limited data and generalize effectively to new datasets, justifying the use of graphs for semantically representing heap structures.

We introduced two classifiers: one predicts the presence of keys in the heap, and the other identifies the starting node for the agent's exploration, both aimed at simplifying the agent's tasks. The Deep Reinforcement Learning agent was trained to navigate these graphs and extract SSH keys successfully. Despite the limited training data, our method demonstrated high effectiveness and adaptability, extracting keys from previously unseen heap dumps with impressive accuracy and independently identifying and extracting multiple keys from a single graph.

However, limitations were observed, including the extensive training required to achieve high accuracy and concerns about generalization across different versions. The training and testing datasets were derived from the same versions, raising the possibility of overfitting to these specific graphs and not generalizing well to new versions.

Our findings suggest several avenues for future research, such as integrating more sophisticated graph neural networks or advanced reinforcement learning algorithms. Techniques like Beam Search [BeamSearch] and Monte Carlo Tree Search [Świ+21] could also be explored to enhance performance.

In conclusion, this thesis demonstrates the feasibility of using graphs and Deep Reinforcement Learning for extracting SSH keys from heap dumps, showing promise as a tool for digital forensics experts. With further development, this method could become an invaluable asset in the field.

Bibliography

[BeamSearch] *AI | Search Algorithms | BEAM Search | Codecademy — codecademy.com.* URL: https://www.codecademy.com/resources/docs/ai/search-algorithms/beam-search (cit. on p. 70).

[And+17] Marcin Andrychowicz et al. "Hindsight Experience Replay". In: *Advances in Neural Information Processing Systems* 2017-December (July 2017), pp. 5049–5059. ISSN: 10495258. URL: https://arxiv.org/abs/1707.01495v3 (cit. on p. 57).

[BGR19] Liu Binxiang, Zhao Gang, and Sun Ruoying. "A deep reinforcement learning malware detection method based on PE feature distribution". In: *Proceedings - 2019 6th International Conference on Information Science and Control Engineering, ICISCE 2019* (Dec. 2019), pp. 23–27. DOI: 10.1109/ICISCE48695.2019.00014 (cit. on p. 2).

[Fel+22] Christofer Fellicious et al. "SmartKex: Machine Learning Assisted SSH Keys Extraction From The Heap Dump". In: (2022). URL: https://github.com/smartvmi/Smart-and-Naive-SSH-Key-Extraction (cit. on pp. 7, 16, 20, 64).

[OSSHPort] *GitHub - openssh/openssh-portable: Portable OpenSSH — github.com.* URL: https://github.com/openssh/openssh-portable (cit. on pp. 6, 7).

[GNN] *GNN.* URL: https://medium.com/the-modern-scientist/graph-neural-networks-series-part-1-an-introduction-49a88941f888 (cit. on p. 13).

[Haa+18] Tuomas Haarnoja et al. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *35th International Conference on Machine Learning, ICML 2018* 5 (Jan. 2018), pp. 2976–2989. URL: https://arxiv.org/abs/1801.01290v2 (cit. on p. 60).

[Haa+24] Tuomas Haarnoja et al. "Learning agile soccer skills for a bipedal robot with deep reinforcement learning". In: *Science Robotics* 9 (89 Apr. 2024). ISSN: 2470-9476. DOI: 10.1126/scirobotics.adi8022. URL: https://www.science.org/doi/10.1126/scirobotics.adi8022 (cit. on p. 2).

[HYL17] William L. Hamilton, Rex Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *Advances in Neural Information Processing Systems* 2017-December (June 2017), pp. 1025–1035. ISSN: 10495258. URL: http://arxiv.org/abs/1706.02216 (cit. on p. 14).

[HGS16] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 30 (1 Mar. 2016), pp. 2094–2100. ISSN: 2374-3468. DOI: 10.1609/aaai.v30i1.10295. URL: https://ojs.aaai.org/index.php/AAAI/article/view/10295 (cit. on p. 49).

[Heap] *Heap*. URL: https://www.geeksforgeeks.org/what-is-a-memory-heap/ (cit. on p. 6).

[SumTree] *Introduction to Sum Tree | Fcode Labs — fcodelabs.com*. URL: https://www.fcodelabs.com/blogs/introduction-to-sum-tree (cit. on p. 49).

[Keras] *Keras: Deep Learning for humans*. URL: https://keras.io/ (cit. on p. 24).

[KW16] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings* (Sept. 2016). URL: http://arxiv.org/abs/1609.02907 (cit. on p. 14).

[Kor24] Ezgi Korkmaz. "A Survey Analyzing Generalization in Deep Reinforcement Learning". In: (Jan. 2024). URL: http://arxiv.org/abs/2401.02349 (cit. on p. 18).

[Lam+22] Remi Lam et al. "GraphCast: Learning skillful medium-range global weather forecasting". In: *arXiv* (Dec. 2022), p. 2212.12794. URL: https://arxiv.org/abs/2212.12794v2 (cit. on p. 13).

[Lil+15] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning". In: *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings* (Sept. 2015). URL: http://arxiv.org/abs/1509.02971 (cit. on p. 17).

[MallocGit] *lsploits/glibc/malloc/malloc.c at master · sploitfun/lsploits — github.com*. URL: https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L1108 (cit. on p. 26).

[Malloc] *MallocInternals - glibc wiki*. URL: https://sourceware.org/glibc/wiki/MallocInternals (cit. on p. 26).

[Mni+13] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (Dec. 2013). URL: http://arxiv.org/abs/1312.5602 (cit. on pp. 11, 17, 18).

[Mni+16] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *33rd International Conference on Machine Learning, ICML 2016* 4 (Feb. 2016), pp. 2850–2869. URL: https://arxiv.org/abs/1602.01783v2 (cit. on p. 17).

[NetworkX] *NetworkX — NetworkX documentation*. URL: https://networkx.org/ (cit. on p. 24).

[NIST2024] *NVD - CVE-2024-3094 — nvd.nist.gov*. URL: https://nvd.nist.gov/vuln/detail/CVE-2024-3094 (cit. on p. 1).

[OpenSSH] *OpenSSH*. URL: https://www.openssh.com/ (cit. on pp. 6, 34).

[OSB] *oss-security - backdoor in upstream xz/liblzma leading to ssh server compromise — openwall.com*. URL: https://www.openwall.com/lists/oss-security/2024/03/29/4 (cit. on p. 1).

[Oversmoothing] *Over-smoothing issue in graph neural network | by Anas AIT AOMAR | Towards Data Science*. URL: https://towardsdatascience.com/over-smoothing-issue-in-graph-neural-network-bddc8fbc2472 (cit. on p. 52).

[PyGDoc] *PyG Documentation — pytorch_geometric documentation*. URL: https://pytorch-geometric.readthedocs.io/en/latest/ (cit. on p. 24).

[PyG] *pyg-team/pytorch_geometric: Graph Neural Network Library for PyTorch*. URL: https://github.com/pyg-team/pytorch_geometric (cit. on p. 24).

[PyTorch] *PyTorch*. URL: https://pytorch.org/ (cit. on p. 24).

[PyTorchDoc] *PyTorch documentation — PyTorch 2.2 documentation*. URL: https://pytorch.org/docs/stable/index.html (cit. on p. 24).

[RFC4252] *RFC4252*. URL: https://www.rfc-editor.org/rfc/rfc4252 (cit. on p. 5).

[RFC4253] *RFC4253*. URL: https://www.rfc-editor.org/rfc/rfc4253 (cit. on p. 5).

[Rust] *Rust Programming Language*. URL: https://www.rust-lang.org/ (cit. on p. 23).

[Sch+15] Tom Schaul et al. "Prioritized Experience Replay". In: (Nov. 2015). URL: http://arxiv.org/abs/1511.05952 (cit. on p. 48).

[sklearn] *scikit-learn: machine learning in Python ; scikit-learn 1.4.2 documentation — scikit-learn.org*. URL: https://scikit-learn.org/stable/index.html (cit. on pp. 24, 38).

[HSA] *Security Alert: Potential SSH Backdoor Via Liblzma — hackaday.com*. URL: https://hackaday.com/2024/03/29/security-alert-potential-ssh-backdoor-via-liblzma/ (cit. on p. 1).

[SR22] 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). ISSN: 26662817. DOI: 10.1016/J.FSIDI.2022.301337 (cit. on pp. 7, 16).

[SSR23] Mohit Sewak, Sanjay K. Sahay, and Hemant Rathore. "Deep Reinforcement Learning in the Advanced Cybersecurity Threat Detection and Protection". In: *Information Systems Frontiers* 25 (2 Apr. 2023), pp. 589–611. ISSN: 15729419. DOI: 10.1007/S10796-022-10333-X/TABLES/3. URL: https://link.springer.com/article/10.1007/s10796-022-10333-x (cit. on p. 2).

[Sil+16] David Silver et al. "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529 (7587 Jan. 2016). ISSN: 1476-4687. DOI: 10.1038/nature16961. URL: http://www.ncbi.nlm.nih.gov/pubmed/26819042 (cit. on pp. 2, 17).

[Świ+21] Maciej Świechowski et al. "Monte Carlo Tree Search: A Review of Recent Modifications and Applications". In: *Artificial Intelligence Review* 56 (3 Mar. 2021), pp. 2497–2562. DOI: 10.1007/s10462-022-10228-y. URL: http://arxiv.org/abs/2103.04931%20http://dx.doi.org/10.1007/s10462-022-10228-y (cit. on pp. 17, 70).

[TAR18] Benjamin Taubmann, Omar Alabduljaleel, and Hans P. Reiser. "DroidKex: Fast extraction of ephemeral TLS keys from the memory of android apps". In: *Proceedings of the Digital Forensic Research Conference, DFRWS 2018 USA* (2018), S67–S76. ISSN: 17422876. DOI: 10.1016/J.DIIN.2018.04.013 (cit. on p. 16).

[Tau+16] Benjamin Taubmann et al. "Tlskex: Harnessing virtual machine introspection for decrypting TLS communication". In: *DFRWS 2016 EU - Proceedings of the 3rd Annual DFRWS Europe* (2016), S114–S123. ISSN: 17422876. DOI: 10.1016/J.DIIN.2016.01.014 (cit. on pp. 15, 16).

[TensorFlow] *TensorFlow*. URL: https://www.tensorflow.org/?hl=fr (cit. on p. 24).

[GraphML] *The GraphML File Format — graphml.graphdrawing.org*. URL: http://graphml.graphdrawing.org/ (cit. on p. 27).

[PyGData] *torch_geometric.data.Data pytorch_geometric documentation — pytorch-geometric.readthedocs.io*. URL: https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.data.Data.html (cit. on p. 30).

[Gv2Conv] *torch_geometric.nn.conv.GATv2Conv — pytorch_geometric documentation*. URL: https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.conv.GATv2Conv.html (cit. on pp. 34, 46).

[GMP] *torch_geometric.nn.pool.global_mean_pool — pytorch_geometric documentation*. URL: https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.pool.global_mean_pool.html (cit. on p. 46).

[Vel+17] Petar Veličković et al. "Graph Attention Networks". In: *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings* (Oct. 2017), pp. 39–41. DOI: 10.1007/978-3-031-01587-8_7. URL: http://arxiv.org/abs/1710.10903 (cit. on p. 14).

[Wat89] C. Watkins. "Learning from delayed rewards". In: (1989) (cit. on p. 9).

[Python] *Welcome to Python.org*. URL: https://www.python.org/ (cit. on p. 23).

[AWK] *What we know about the xz Utils backdoor that almost infected the world — arstechnica.com*. URL: https://arstechnica.com/security/2024/04/what-we-know-about-the-xz-utils-backdoor-that-almost-infected-the-world/ (cit. on p. 1).

[Wireshark] *Wireshark · Go Deep*. URL: https://www.wireshark.org/ (cit. on p. 16).

[Wu+23] Xinyi Wu et al. "Demystifying Oversmoothing in Attention-Based Graph Neural Networks". In: (May 2023). URL: https://arxiv.org/abs/2305.16102v3 (cit. on p. 52).

[YL06] T. Ylonen and C. Lonvick. "The Secure Shell (SSH) Protocol Architecture". In: (Jan. 2006). Ed. by C. Lonvick. ISSN: 2070-1721. DOI: 10.17487/RFC4251. URL: https://www.rfc-editor.org/info/rfc4251 (cit. on pp. 4, 33, 34).

[ZZR19] Zihao Zhang, Stefan Zohren, and Stephen Roberts. "Deep Reinforcement Learning for Trading". In: *The Journal of Financial Data Science* 2 (2 Nov. 2019), pp. 25–40. ISSN: 2640-3943. DOI: 10.3905/

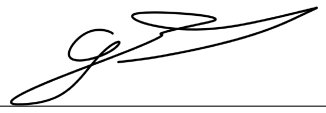
jfds.2020.1.030. URL: http://arxiv.org/abs/1911.10107 (cit. on p. 2).

[Zhu+17] Pengfei Zhu et al. "On Improving Deep Reinforcement Learning for POMDPs". In: (Apr. 2017). URL: http://arxiv.org/abs/1704.07978 (cit. on p. 40).

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, April 18, 2024



Cyril GOMES