Large Scale Heap Dump Embedding for Machine Learning: Predicting OpenSSH Key Locations

Florian Rascoussier, Lise Lahoche

Large Scale Heap Dump Embedding for Machine Learning: Predicting OpenSSH Key Locations

Florian Rascoussier[1,2,3] [0009−0005−3253−9814] and
Clément Lahoche[1,2,4] [0009−0007−7687−5419]

[1] Universität Passau, Innstr. 43, 94032 Passau, Germany
[2] INSA Lyon, 20 Avenue Albert Einstein, 69100 Villeurbanne, France
[3] florian.rascoussier@insa-lyon.fr
IMT Atlantique, Technopôle Brest-Iroise, 29238 Brest, France
[4] INRIA Rennes, 263 Av. Général Leclerc, 35042 Rennes, France
clement.lahoche@inria.info

Abstract. With the evolving landscape of cybersecurity, forensic analysis has become increasingly pivotal, especially with the integration of machine learning (ML) techniques. However, the use of ML in cybersecurity, and especially in heap dump analysis is still in its infancy, both due to the lack of quality datasets and the difficulty of processing large-scale heap dumps collections. Another complex task lies in the transition from raw byte heap dump data into dense vector representations, or embeddings, that can be used with ML models. This paper addresses these challenges by introducing a novel methodology and the Mem2Graph tool for processing large-scale heap dumps collections. This method has been introduced while developing a novel approach to enhance the detection of session keys in OpenSSH heap dumps. Such a novel approach has significantly advanced the state of the art in predicting the location of keys in OpenSSH heap dumps. Importantly, it paves the way for automated ML applications that leverage the structure and embeddings from reconstructed memory graphs, opening new frontiers in both cybersecurity and data science.

Keywords: SSH keys · Heap dumps · Embeddings · Machine learning · Knowledge graphs · Graph Convolutional Networks · Memory Graphs

1 Introduction

Encrypted communications have become ubiquitous in maintaining privacy and security over the internet. Among these, Secure Shell (SSH) is pivotal for encrypted remote server access, but this encryption poses challenges for forensic analysis [18]. Traditional methods for obtaining decrypted SSH traffic, such as active man-in-the-middle attacks or binary manipulation, have

limitations in terms of detectability and ethical implications [10]. A novel approach, as explored in SSHKex [18], leverages virtual machine introspection (VMI) to extract SSH session keys from memory, thereby decrypting SSH traffic with minimal alteration of server binaries that is nearly undetectable by users. However, the method introduced with SSHKex is highly specific and requires a large amount of manual work, and a precise match of program versions. This makes it difficult to apply this approach to a broader set of versions or to extend it to other programs. New research proposed by the SmartKex team demonstrates the interest of machine learning techniques for the detection of session keys in OpenSSH heap dumps [6].

Contributions This paper builds on the foundation laid by SSHKex and SmartKex, introducing a twofold advancement in the realm of memory forensics. The main contributions are:

- A general methodology and open source tool (Mem2Graph[5]) capable of processing large-scale heap dump dataset for advanced exploration and ML model training. This tool not only segregates corrupt files but is also both capable of constructing memory graphs and semantic-rich vector representation of heap dumps.
- The method directly serves as a novel approach to enhance the detection of session keys in OpenSSH heap dumps. Our approach surpasses the existing state of the art, enabling direct prediction of session key locations (direct address) within the heap, a significant leap forward compared to previous technique. The method is also much more generic and could be applied to other programs.

The following provides a methodology for processing heap dump datasets, followed by a novel approach for enhancing SSH key detection in heap dumps. Finally, findings and discussions will be presented.

2 Background

Memory forensics, a subfield of digital forensics in cybersecurity, focuses on the analysis of volatile data in a computer's memory. This field is critical for investigating cyberattacks, as volatile data contains information like running processes, open network connections and associated encryption keys. Being able to analyze this data is crucial for identifying malicious activity and gathering evidence of cyberattacks. Programs like OpenSSH, an open-source implementation of SSH that is widely used across various platforms, play a significant role in secure communication. It relies on volatile data to store its session keys and other sensitive information.

[5] Open Source: https://github.com/passau-masterarbeit-2023/mem2graph

2.1 Heap Dumps in Memory Forensic

In program memory, data structures are typically stored in the heap, a region of memory used for dynamic memory allocation. Heap dumps are snapshots of the heap memory at a specific point in time. Their analysis is influenced by factors such as the version of the program, operating system, and architecture. Understanding heap dumps involves parsing these snapshots to reveal intricate connections between data structures and pointers. To do so, minimum knowledge is required, such as the heap start address, the endianess and the memory layout. Below is an example of content of a raw memory dump file:

```
--offset: ------value in hex: --ASCII:
00000048: 0000 0100 0000 0001 ........
00000050: 8022 1a3a 3456 0000 ."."4V..
00000058: 007f 1a3a 3456 0000 ...:4V..
```

Listing 1.1. 8 bytes per line visualization of a piece of Hex Dump from *Training/basic/V_7_8_P1/16/5070-1643978841-heap.raw*.

The following terminology will be used throughout the paper:

- **memory graph:** It refers to a directed graph that represents the memory of a heap dump file. The memory graph is defined in sec. 4.2 as the main data structure used for the embedding step.
- **block:** A sequence of 8 bytes. A heap dump file is composed of a sequence of blocks where each has an address and a value.
- **chunk:** A chunk is a sequence of blocks. This concepts directly comes from the `malloc` function in C. At its core, a chunk has a user data body composed of blocks and a malloc header block.

2.2 OpenSSH

OpenSSH, an open-source implementation of SSH, plays a significant role in secure communication. OpenSSH uses different algorithms and hash functions, for securing data, with a focus on forward secrecy. SSH key management involves a key exchange process using algorithms to establish a shared secret for encryption, with up to 6 keys per session (2 for encryption, 2 for integrity checks, and 2 for initialization vectors) [18]. This process results in multiple session keys stored in the heap.

3 Related Work

This section outlines the advancements in OpenSSH key extraction.

3.1 SSHKex

SSHKex, a project utilizing VMI, focuses on extracting SSH keys and decrypting SSH traffic non-intrusively. It requires detailed knowledge of SSH implementation data structures and function tracing in OpenSSH, leveraging its open source implementation. It uses breakpoints in specific functions to extract keys, ensuring stealthiness and evidence integrity. Communications are also captured between the targeted client and server for later decryption using the extracted keys [18].

3.2 SmartKex

SmartKex extends SSHKex's work, aiming to automate SSH key extraction from heap memory dumps using machine learning. Unlike SSHKex, it does not directly obtain key locations, because it has no prior knowledge of the program implementation. Instead, it identifies high-entropy heap sections likely to contain keys. This identification using a RandomForest model is followed by a brute-force approach for actual key extraction [6].

3.3 Research gap and objectives

Previous works have focused on extracting keys from OpenSSH heap dumps using either implementation knowledge (SSHKex) or ML enhanced entropy-filtered bruteforce (SmartKex), but there is a lack of research on the preprocessing of heap dumps for ML techniques. This paper aims to bridge this gap by introducing a novel methodology and the Mem2Graph tool that covers different approaches for heap dump exploration and embedding, with a focus on ML and deep learning applications applied to OpenSSH key prediction. This new approach is directly inspired from recent advances in the field of knowledge graphs and graph neural networks (GCNs). The Mem2Graph tool has been designed with genericity in mind, and is available on GitHub[6].

4 Methodology

This section dives into the methodology for Heap Dump Processing, Memory Graph Construction as well as the different possible embeddings.

4.1 Heap Dump Processing

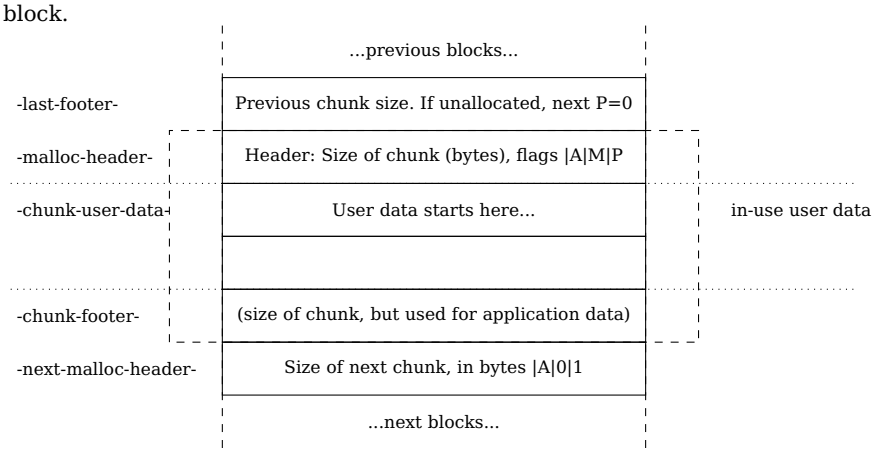
The present work relies on a dataset of heap dump files from OpenSSH sessions, generated on a x86_64 architecture, Linux (Debian)[7] OS with Debian

[6] https://github.com/passau-masterarbeit-2023/mem2graph

[7] According to Hans Reiser: Linux debian10 4.19.0-16-amd64 #1 SMP Debian 4.19.181-1 (2021-03-19) x86_64 GNU/Linux

GLIBC 2.28-10. Since OpenSSH implementation is in C, it relies on the *malloc* function from the GLIBC library to allocate memory on the heap. *malloc* uses a boundary tag method to manage in-use and remaining free chunks of memory. Each chunk contains metadata that helps in the allocation and deallocation of memory [4,7]. Those components can be used to retrieve information about the chunk like its size, its status (in-use or free), and the address of the next chunk [2,4]. Figure 1 is a diagram of an allocated chunk in GLIBC [7]:

Fig. 1. Diagram of an allocated chunk in GLIBC 2.28 [7]. Each rectangle represents a block.



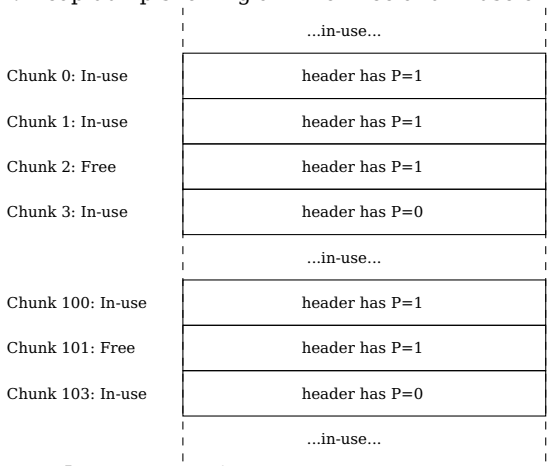
Assumptions We consider the following assumptions for each heap dump:

- **Same program origin:** The embeddings are developed to capture information about the memory of a specific program, here OpenSSH.
- **Heap dump file size:** A multiple of 8 bytes as a sequence of blocks.
- **Chunk chaining:** All the heap dump files have been generated using the same `malloc` implementation for memory pattern consistency. Each file is expected to start by a first allocated in-use chunk. Using information from chained malloc headers, exploration of the heap dump file is possible [4]. Note that only the malloc header is really needed, meaning that this method can be adapted to other allocation functions that rely on the same boundary tag technique.
- **Dataset annotations:** Additional annotations are needed for the exploration and embedding of heap dump files:
  - **Heap start address:** Required to compute the real address of blocks in the heap. This is the address of the first block of the heap dump file in the program memory, which needs to be stored when generating the heap dump file.
  - **Endianess:** Needed to correctly interpret the value of each block, and especially pointers.

- **Other annotations:** Depending on the program and task to perform on the dataset, additional annotations might be needed. In the case of OpenSSH key prediction, the address of each session key is required for training.

A first cleaning step is needed to ensure that all the heap dump files are valid by following the assumptions stated above. A valid heap dump can thus be parsed by following the chaining of malloc headers. The following diagram shows the typical structure of a heap dump file:

Fig. 2. Heap dump showing a mix of free and in-use chunks.



4.2 Memory Graph construction

The memory graph stands as the principal data structure employed in the embedding phase. Its design is fundamentally driven by the objective to encapsulate the memory architecture inherent in heap dump files for ML purposes. It is pivotal in addressing the challenges posed by the dataset imbalanceness and the curse of dimensionality (see sec. 5.1). The memory graph is a directed graph drawing direct inspiration from the principles of Knowledge Graph (KG) theory [11]. There are 2 types of such graphs:

- **Block graph:** Composed of nodes that represent blocks and edges that represent pointers from one block to another, or chunk ownership relationship from a malloc header considered as a chunk root node to its block nodes contained within the chunk. This second relationship is directly inspired from the `rdf:Bag` container of the Resource Description Framework (RDF) ontology, which is used to model unsorted collections of resources in a KG[3, p69]. This graph contains all the information of the heap dump file, but is very large and thus difficult to process.
- **Chunk graph:** Composed of nodes that represent chunks and edges that represent pointers from one chunk to another. It is several orders of

magnitude smaller than the block graph, and is thus much more efficient to process. In the following, we will focus on this type.

This graph is constructed for each heap dump by a dual parsing process:

1. **Chunk parsing**: The first parsing of the heap dump follows the chunk chaining assumption (see 4.1) to construct all the chunk nodes.
2. **Pointer parsing**: The second one consists at iterating over all the chunk nodes and parsing their blocks to construct the pointer edges.

Note that, for ML purposes, the addition of block or chunk annotation is possible in a third stage. A block is considered a pointer if its value is a valid address within the heap dump address range. This is based on the fact that, for any block of 8 bytes, and a typical heap dump containing 100 000 blocks (as per our dataset), the probability of a random block to be a pointer is very low. This stays true even for bigger heap dumps, as this probability is approximately equal to 5×10^{-15} (it is equal to $n \div 2^{\text{k} \times 8 \text{ bits}}$ where n is the number of blocks in the heap dump (the range) which is generally around $100\,000$, and k is the block size which is equal to 8). Once the memory graph is constructed, it can be saved in different formats for further processing. Mem2Graph supports the popular human-readable DOT format [5].

4.3 Embeddings

Embeddings are a pivotal concept in data representation, particularly in machine learning. A feature, in this context, refers to measurable attributes or characteristics of phenomena. An embedding is defined as a fixed-size feature vector representation of a high-dimensional object, serving the purpose of capturing complex relationships between those objects. This is especially useful in scenarios such as Natural Language Processing (NLP) or graph analysis, where embeddings involve mapping discrete objects like words or nodes to vectors of values in a lower-dimensional space, thereby simplifying and enriching their representation [8]. Embeddings can be manually engineered (feature engineering) or automatically learned using deep learning techniques. The following section describes 2 different embedding strategies compared in this work.

Embedding from Bytes Those embeddings are directly obtained from the content of each chunk which can vary in size:

- Embeddings based on feature engineering from chunk memory graph:
  - **Padding Method**: Add zeros at the end of the chunk to match the size of the largest chunk. However, this approach results in very large embeddings (65536 features) and is not efficient.

  - **Trimming Method**: Cut down larger chunks to match the size of smaller ones but the size of the cut needs to be determined.
  - **Statistical Method**: Compute statistics on the bytes of the chunk, including the mean, standard deviation, MAD (Mean Absolute Deviation), Skewness [21], Kurtosis [21], Shannon entropy, and N-grams[8].
  - Those embedding are completed with 4 common features: the chunk size in bytes, the number of pointers inside the chunk, the number of other blocks inside the chunk and the chunk position in the dump.
- Automated embeddings using Deep Learning NLP models:
  - Consider the byte content of the chunk as sentences and combinations of bytes as words.
  - Two state-of-the-art, advanced NLP models have been tested: **Word2Vec** [13] and **Transformers** [20].

Embedding from graph Graph embedding techniques aim to map nodes and edges in a graph to vectors in a low-dimensional space [11]. The primary goal is to preserve the graph's structural properties, such as node connectivity and community structure, in the embedded space. Many embeddings have been developed [11], like Node2Vec [9], DeepWalk [15], LINE [19], Graph Factorization, etc. **Node2Vec** has been chosen following the need to capture the specificities (see 4.1) and complex local structures from memory graphs for later ML applications. While its versatility is a strong point, it comes at the cost of increased resource consumption due to the introduction of several hyperparameters controlling the Random Walks.

5 Application to OpenSSH Session Key Prediction

The following describes the application of the methodology to the problem of OpenSSH session key prediction.

5.1 Dataset

The dataset provided by the SmartKex team[9] [6], and refined by the authors[10] following the assumptions (see 4.1), is composed of 26202 valid heap dump binary files from OpenSSH[11] together with their JSON annotation file. The task of predicting the addresses withing each dump is highly imbalanced. Each heap dump is composed of around 100 000 blocks, and only 6 keys, meaning 6 addresses are positively labelled. Thus, the imbalance ratio

[8] Combination of N bits
[9] Available on Zenodo: https://zenodo.org/records/6537904
[10] Available on Zenodo: https://zenodo.org/records/10514199
[11] OpenSSH on different versions (from V6.0 to V8.8). Architecture, see 4.1

is around 1:17000 positive blocks. However, an important discovery is that the keys are always located at the beginning of their parent chunk. This allows to reconsider the problem as a binary chunk classification, where the goal is to predict if a chunk contains a key or not. This improves the balancing of the dataset since every dump is only composed of around 1000 chunks. So the new ratio is around 1:170 positive chunks. Chunk filtering strategies have also been tested, like entropy filtering (see 3.2), chunk size (16, 32, 64 bytes depending on the size of keys) or in-used chunk filtering.

5.2 Embeddings and model comparison for key prediction

Comparing the embeddings is not a trivial task. Indeed, the embeddings from bytes[12] are much more memory efficient than the embedding from graph. The limiting factor being memory, 2 experiments were launched in parallel on 2 servers with 512 GB of RAM. The first experiment is done on the whole dataset and compares the different embeddings from bytes. Chunk filtering techniques have been used to reduce the imbalance ratio, as well as a feature selection method (Pearson correlation matrix [1]) to ensure a subset of 8 fair features for each compared embedding. The RandomForest model is used for comparison with previous work [6]. The second experiment is done on small subsets of the dataset and compares all the embeddings, alone or in combination. It was also used to test other ML and Deep Learning (DL) models (RandomForest, SGDClassifier, LogisticRegression from Scikit-Learn [14]) and some custom GCN models:

- **Very simple GCN**: minimalist model with one GCN layer, followed by one fully connected layer. Convolution layer takes input features and transforms them into a 16-dimensional space. A ReLU activation function is applied to the output to introduce non-linearity in the model.
- **Simple GCN**: Similar but with 2 GCN layers.
- **First GCN Model**: First implemented version with 2 GCNs and 3 fully connected layers.
- **GCN with dropout**: Same as previous, with additional dropout (rate of 0.5 after each ReLU activation) and batch normalization layers to make the model both more robust and faster to train.
- **Advanced GCN**: Similar to previous but with 3 GCN layers (32, 64 and 128 dimensions).

6 Results and Discussion

The following section regroups the results of the 2 experiments conducted. The focus is on the recall because the objective is to detect all the potential

[12] **Note:** The trimming embedding considers the first 12 bytes of the chunk since the smallest key is 12-byte long.

keys to be used later with a bruteforce post-validation. For each series, training and evaluation subsets are used, but no cross-validation (CV) is done due to a range of factors: the added computation cost, the dataset limited number of positives, the fact that CV is not recommended for feature selection, or the fact that many instances are trained and need to be comparable [16].

6.1 Previous results from SmartKex

The following table 1 is extracted from the SmartKex paper [6]:

Table 1. Results from SmartKex [6].

Classifier	Accuracy	Precision	Recall	F1-score
High Precision	0.9975	0.9317	0.8437	0.8855
High Recall	0.9906	0.5553	0.9962	0.7131
stacked	0.9956	0.7609	0.9160	0.8313
window to address for comparison	0.0622	0.0475	0.0572	0.0519

An important caveat about those results is that the predictions are done on raw "128-byte slice of data" [6], and thus require bruteforce for key extraction whereas the current work predicts directly the address of keys, and thus is 16 times more precise from start, by design.

6.2 Results of first experiments on full dataset

The first series of experiments was focused on exploring features and embedding models and their impact on key prediction. As such, this series was conducted using the whole cleaned dataset. It used 24699 heap dumps from `Training` (20964) and `Validation` (3735), with a ratio of 0.178 between training and evaluation steps. It also included a timeout (5600s), memory limit (around 512 GB), and a limited hyperparameter grid-search (see 2 and 3) to match the performance imperative of the experimental setup.

Table 2. Tested hyperparameters for the instances of the Transformer model.

Word size	16	16	8	8	16	16	8	8
Embedding dimension	8	16	8	16	8	16	8	16
Transformer units	2	2	2	2	4	4	4	4
Num heads	2	2	2	2	4	4	4	4
Num layers	2	2	2	2	4	4	4	4
Dropout rate	0.1	0.1	0.1	0.1	0.3	0.3	0.3	0.3

According to table 4, the best embedding for key prediction on recall is the *trimming method* with *Random undersampling* filtering, on the whole dataset achieving great improvement from results provided by SmartKex (1

Table 3. Tested hyperparameters for the instances of the Word2Vec model.

Embedding dimension	8	8	8	8	16	16	16	16	100	100	100	100
Window character size	8	8	16	16	8	8	16	16	8	8	16	16
Word size	2	4	2	4	2	4	2	4	2	4	2	4

Table 4. Byte embedding performance results

Methods	**Filter**	**Precision**	**Recall**	**Accuracy**	**f1-score**
trimming method	Random undersampling	0.7391	1.0000	0.9985	0.8499
	Chunk size filter	0.8354	0.9995	0.9990	0.9101
	Entropy filter	0.9176	0.9977	0.9853	0.9559
	Chunk size and entropy filter	0.8972	0.9999	0.9735	0.9458
Statistical method	Random undersampling	Out Of Memory (OOM)			
	Chunk size filter				
	Entropy filter	0.9686	0.9980	0.9945	0.9831
	Chunk size and entropy filter	0.9335	0.9902	0.9811	0.9610
Best DL instance	Chunk size and entropy filter	0.9511	0.9992	0.9879	0.9746

versus 0.0572). Note that the same embedding with both *chunk size* and *entropy* filtering is just one missed key away from the previous, but has a much better precision. This is maybe due to noise. The best F1-Score is the *Statistical method* with *Entropy* filtering (0.9831 vs 0.0519). The DL instances (Transformers and Word2Vec) have always lesser results compared to the other embeddings. This is probably a consequence of the limited hyperparameter search. The statistical method has run OOM because of its size (280 features due to the N-Gram metric).

6.3 Results of second experiments on small subset of the dataset

This series of experiments was conducted on an increasing subset of the cleaned dataset, containing 16, 32 and 64 heap dumps with a ratio of 0.2 for evaluation over training. It totals more than 400h of in-parallel computation time on an 80-core server with 512 GB of RAM. The goal was to compare the different embeddings and models on a small dataset, especially the Node2Vec and GCN models, which are computationally and memory-intensive. A more extensive hyperparameter search was also conducted on both Node2Vec and classification models[13].

In total, the experiments including hyperparameter search lead to 13427 different models trained and tested, with a total *Node2Vec* embedding time of more than 136 days of synchronous time, or 29.86 times higher than the

[13] Raw CSV of results with hyperparameters can be directly consulted here: https://github.com/passau-masterarbeit-2023/masterarbeit_report_onyr/blob/main/src/results/csv/concatenated_csv/concatenated_raw_result_v2.csv

Table 5. Best instance for each model, with respect to f1 score.

Model	Accuracy	Precision	Recall	F1 Score	Embedding
sgd-classifier	0.9924	0.4615	1.0000	0.6316	node2vec
very-simple-gcn	0.9946	0.6000	0.5000	0.5455	node2vec
first-gcn	0.9935	0.5000	0.5000	0.5000	node2vec
simple-gcn	0.9935	0.5000	0.5000	0.5000	node2vec
logistic-regression	0.9912	0.3333	0.5000	0.4000	node2vec
gcn-with-dropout	0.9858	0.2110	0.7667	0.3309	node2vec
advanced-gcn	0.9898	0.2097	0.4333	0.2826	node2vec
random-forest	0.9984	1.0000	0.0833	0.1538	trimming

training time. Since the focus is on graph embedding, the chunk memory graphs from heap dumps need to be complete meaning that it was not possible to use chunk filtering. According to table 5, *Node2Vec* graph embedding is clearly the best for all but *RandomForest* where the first series of experiments showed excellent results for this model on byte embeddings like *trimming*.

Table 6. Best model instance for each metric.

Metric	Model	Accuracy	Precision	Recall	F1 Score	Embedding
accuracy	random-forest	0.9984	1.0000	0.0833	0.1538	trimming
precision	logistic-regression	0.9944	1.0000	0.0417	0.0800	node2vec
recall	first-gcn	0.9826	0.2727	1.0000	0.4286	node2vec
f1 score	sgd-classifier	0.9924	0.4615	1.0000	0.6316	node2vec

With table 6, the best classification model instance for optimal recall is given with several models like *SGDClassifier* and *FirstGCN* ; and for precision, with models like *RandomForest* and *LogisticRegression*. The best overall model (with highest F1-Score) is *SGDClassifier*, but its score of 0.6316 is much worse than the ones obtained with the full dataset in the first series of experiments (0.9831 with Statistical and Entropy filter). More complex GCN models tend to have decreasing precision but improved recall. This is probably due to the fact that the dataset is very small, and the models are overfitting. The best GCN model is the simplest one, with a F1-Score of 0.5455 which is worse than expected.

6.4 Discussing limitations

Several limitations with this work needs to be considered. First, no time comparison between the different embeddings is provided, but *Node2Vec* is clearly several order of magnitude slower during the training phase. Second, it was not possible to test all state-of-the-art graph embeddings and classifiers due to time and resources constraints. Third, there was no GPU computing because of VRAM limitations.

7 Conclusion

Within memory forensic analysis, heap dumps are a rich source of information. However, they are difficult to process, and there is no standard way of representing them. Machine learning techniques have been shown to be useful for heap dump analysis, but they require a dense representation of the heap dumps. This report demonstrates that memory-graph representation of heap dumps can be efficient for providing byte and graph embeddings. The authors have introduced the Mem2Graph tool for this precise task, as well as a methodology for dealing with heap dump files for ML purposes. Simple classification models like RandomForest and SGDClassifier can be trained very efficiently and appear to be better than more complex models like GCNs for key prediction. Our experimental results show much better results than previous state-of-the-art work on key prediction from heap dumps. This paper opens the door to further research on the use of graph modelization from heap dump for ML tasks.

Future Work Many avenues for future work are possible. These include conducting a more comprehensive comparison of different embedding and classification techniques, analyzing the impact of different C libraries and programming languages on memory layout, and enhancing Mem2Graph to support a wider range of graph formats and heap dump types. The authors have also tried some clustering techniques on chunks, but those research would need to be pushed further. Pursuing this direction could significantly advance the development of a universal machine learning-assisted memory forensics tool for key extraction and other tasks.

Disclosure of Interests. This work is a continuation of the PhDTrack program involving the authors [12,17], Universität Passau and INSA Lyon. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Boddy, R., Smith, G.: Statistical methods in practice: for scientists and technologists. John Wiley & Sons (2009)
2. Daeumer, J., Das, D., Fleury, E.: How does glibc malloc work? (2023), https://reverseengineering.stackexchange.com/questions/15033/how-does-glibc-malloc-work, accessed: 2023-09-25
3. Decker, S., Mitra, P., Melnik, S.: Framework for the semantic web: an rdf tutorial. IEEE Internet Computing **4**(6), 68–73 (2000)
4. Delorie, D., Weimer, F., O'Donell, C., Schwab, A.: Malloc internals (2023), https://sourceware.org/glibc/wiki/MallocInternals, commit: b39f275c, Accessed: 2023-09-25

5. Ellson, J., Gansner, E.R., et al.: Graphviz and dynagraph — static and dynamic graph drawing tools. In: Jünger, M., Mutzel, P. (eds.) Graph Drawing Software, pp. 127–148. Springer Berlin Heidelberg (2004)
6. Fellicious, C., Sentanoe, S., Granitzer, M., Reiser, H.P.: Smartkex: Machine learning assisted ssh keys extraction from the heap dump (arXiv:2209.05243) (09 2022). https://doi.org/10.48550/arXiv.2209.05243, arXiv:2209.05243 [cs]
7. Gloger, W., Lea, D.: Malloc implementation for multiple threads without lock contention (2001), https://elixir.bootlin.com/glibc/glibc-2.28/source/malloc/malloc.c, version ptmalloc2-20011215, based on VERSION 2.7.0, Sun Mar 11 14:14:06 2001 by Doug Lea. Accessed: 2023-09-22
8. Gomez-Perez, J.M., Denaux, R., Garcia-Silva, A.: Understanding Word Embeddings and Language Models, pp. 17–31. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-44830-1_3
9. Grover, A., Leskovec, J.: node2vec: Scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 855–864 (2016)
10. Hetzler, C., Chen, Z., Khan, T.M.: Analysis of ssh honeypot effectiveness. In: Advances in Information and Communication. pp. 759–782. Springer Nature Switzerland (03 2023). https://doi.org/10.1007/978-3-031-28073-3_51
11. Hogan, A., Blomqvist, E., et al.: Knowledge graphs. ACM Comput. Surv. **54**(4) (7 2021). https://doi.org/10.1145/3447772
12. Lahoche, C.: Structure embeddings for openssh heap dump analysis (2023), https://github.com/passau-masterarbeit-2023/masterarbeit-report-clement/blob/main/report/main.pdf
13. Mikolov, T., Chen, K., et al.: Efficient estimation of word representations in vector space, http://arxiv.org/abs/1301.3781
14. Pedregosa, F., Varoquaux, G., et al.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
15. Perozzi, B., Al-Rfou, R., Skiena, S.: Deepwalk: Online learning of social representations. In: Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 701–710 (2014), https://dl.acm.org/doi/pdf/10.1145/2623330.2623732
16. Rao, R.B., Fung, G., Rosales, R.: On the dangers of cross-validation. an experimental evaluation. In: Proceedings of the 2008 SIAM international conference on data mining. pp. 588–596. SIAM (2008)
17. Rascoussier, F.: Predicting ssh keys in open ssh memorydumps (2023), https://github.com/passau-masterarbeit-2023/masterarbeit_report_onyr/blob/main/report/main.pdf
18. Sentanoe, S., Reiser, H.P.: Sshkex: Leveraging virtual machine introspection for extracting ssh keys and decrypting ssh network traffic. Forensic Science International: Digital Investigation **40**, 301337 (2022). https://doi.org/10.1016/j.fsidi.2022.301337
19. Tang, J., Qu, M., et al.: Line: Large-scale information network embedding. In: Proceedings of the 24th international conference on world wide web. pp. 1067–1077 (2015), https://dl.acm.org/doi/pdf/10.1145/2736277.2741093
20. Vaswani, A., Shazeer, N., et al.: Attention is all you need. Advances in Neural Information Processing Systems **30**, 5998–6008 (2017)
21. Wheeler, D.J.: Problems with skewness and kurtosis. Quality Digest Daily (2011)