Reproducing and Validating Malware Detection through Memory Feature Engineering

Björgvin Freyr Jónsson
Reykjavík University
Reykjavík, Iceland
bjorgvinj17@ru.is

Guðmundur Óli Halldórsson
Reykjavík University
Reykjavík, Iceland
gudmunduroh20@ru.is

ABSTRACT

This paper presents our efforts to reproduce and validate the results of the study "Detecting Obfuscated Malware using Memory Feature Engineering" by Carrier et al. We used their published dataset to reproduce their result and then used a different dataset to validate them. We faced challenges due to format and tool compatibility and found significant discrepancies in the results when applying their methodology to the new dataset.

KEYWORDS

Malware Detection, Machine Learning, Memory Feature Engineering, Paper Result Reproduction, VolMemLyzer, AFF4

1 INTRODUCTION

The paper "Detecting Obfuscated Malware using Memory Feature Engineering" [1] introduced a novel malware detection framework to meet the need for better detection of advanced and obfuscated malware due to the complexity and time requirement to detect manually. The framework leveraged a two-layer stacked ensemble classifier and improvements of an existing memory feature extraction tool called VolMemLyzer [6] by adding 26 new memory-based features to the existing 29 features of the older version. The paper also included a comprehensive dataset called "MalMemAnalysis-2022", which was created with 2916 malware samples of Spyware, Ransomware, and Trojan Horse categories [1].

The evaluation of the model's results was very promising when classifying the obfuscated malware memory samples, but it was 99% in accuracy, precision, recall, and F1-score [1].

Our research aims to reproduce the model and results featured in the paper mentioned above and attempt to validate the results against another dataset [9] of benign and malware memory snapshots. The contribution of this research:

(1) **Reproduction of Original Paper:** We successfully replicated the original paper's study, ensuring the research findings' consistency and reproducibility.
(2) **Validation with New Dataset:** Our project attempted to validate the original model using a different memory snapshot dataset. This step is crucial to evaluate its effectiveness against other categories of malware.
(3) **Development of AFF4 to RAW Format Decoder in Rust:** The main contribution of this research is a decoder to convert AFF4 (Advanced Forensic Format Version 4) memory images to RAW format. In our research, the biggest obstacle was to convert the validation memory images into the raw format that VolMemLyzer [6] or, more accurately, Volatility v2 [10], which VolMemLyzer utilizes in analyzing the memory images. Our efforts to find a tool to do so were unsuccessful. Therefore, making our own was the only option. It could be a critical utility for future work in memory forensics, requiring processing AFF4 memory images in a raw format.
(4) **A minor CSV dataset of pre-processed memory snapshots:** We also provide a small CSV dataset containing 11 benign and 13 malware feature extracted samples from the validation dataset.[9] We used our AFF4 to RAW format decoder and then processed it with VolMemLyzer. The set size is due to time constraints due to the size and scope of this research and the time and space complexities required to process the data in larger chunks.

This paper's structure is as follows. Section 2 provides the necessary background to understand our research. Section 3 goes into the research methodology, details of the two datasets used and their differences, and the tools and challenges we faced, including the design of our AFF4 to RAW converter.

Section 4 presents the results of our research. Section 5 features the discussion section, where we go into the implications of the dataset differences and their effect on the evaluation of the validation dataset[9] using the proposed model. Lastly, section 6 is the paper's conclusion, where we discuss our findings and implications of future work.

2 BACKGROUND

Our research utilizes two primary tools—Volatility and VolMemLyzer—as well as the AFF4 format. This section offers a brief description of these tools and format, providing readers with sufficient information to understand the rest of the paper.

2.1 Volatility

The Volatility Framework, often simply called Volatility, is collection of tools implemented in Python for extracting digital artifacts from volatile memory (RAM) samples [10]. It allows you to extract information such as processes that were running or DLL files that were loaded from a memory dump of a system. It support a number of different versions of Windows, Linux and macOS, although not all commands are supported on every platform.

There are two versions of Volatility that are currently supported, Volatility, which is written in Python 2, and Volatility 3, a complete rewrite that is written in Python 3. In this paper, we will be exclusively using and referring to the older version.

2.2 VolMemLyzer

Volatility Memory Analyzer, or VolMemLyzer, is a Python script that executes multiple commands in volatility on memory samples, and the processes the output from each command to generate a number of different features for each memory sample [6]. The output features from this can then be used by a machine learning algorithm to learn patterns from static memory dumps.

2.3 AFF4

Advanced Forensics Format 4 (AFF4) is a file format for storing digital evidance inside a container [3]. It allows you to have multiple different "objects", where each contains different data, allowing you to store f.ex. both a hard disk dump and a memory dump inside a single AFF4 file.

Usually, a Zip64 container is used to store the AFF4 data. The standard does also allow the data to be stored directly on the filesystem, but during our research, we were unable to find a public implementation of the format that supports that.

AFF4 supports a few different types of "objects" that store data in different ways, but the only one relevant to this paper is the "image" type. It stores data as chunks each compressed with Zlib, and additionally removes gaps (0-byte segments) from the data, which it then stores as metadata alongside each "object".

3 METHODOLOGY

As the aim of this paper is to replicate and validate the results of the paper "Detecting Obfuscated Malware using Memory Feature Engineering", we had to recreate both the pre-processing pipeline and the classifier from the paper, and in this section we will be going over the methods we used and the challenges we faced during that process.

3.1 Datasets

Description of the original dataset and the new dataset used for validation, highlighting the differences in formats (raw vs. AFF4).

We used a pre-processed dataset from the original paper that was created using the VolMemLyzer tool [1]. The dataset includes a total of 58,596 samples. These are evenly split between benign and malicious samples, with each category containing 29,298 instances. The malicious samples in the dataset represent three different types of malware: Ransomware, Spyware, and Trojan Horse.

To validate our model, we used a second dataset comprised of static memory dumps from a Windows 10 system [9]. These dumps included both benign instances and those infected with three distinct types of malicious software: reverse meterpreter shells, Shellter shells, and shells utilizing Hyperion and PEScrambler. The dataset encompasses 4,600 samples in total, with a significant majority of 4,300 being malicious, while the remaining 300 are benign. All samples in this dataset were encoded using the AFF4 format.

The primary difference between these two datasets, aside from one being pre-processed, is that the types of malicious software used is completely different. The training/testing included Ransomware, Spyware, and Trojan Horses, each leaving specific patterns in the system. In contrast, the validation set consists of reverse meterpreter shells, Shellter shells, and shells using Hyperion and PEScrambler. The patterns generated by these different malware categories can vary significantly, which could mean that a model trained on one set may not seamlessly adapt to the specific patterns present in the other.

3.2 Tools and Challenges

Discussion on the challenges faced with VolMemLyzer and the creation of a custom decoder in Rust.

3.2.1 VolMemLyzer and Volatility. The original paper utilized the VolMemLyzer tool for analyzing static memory dumps [1]. However, we encountered several inconsistencies when comparing the VolMemLyzer version available on GitHub with what was described in the paper [6].

Firstly, the GitHub version of VolMemLyzer used more commands in Volatility to generate additional features. However, some of these commands were incompatible with Windows 10, notably the networking commands. As a result, the VolMemLyzer, in its shared state, was not fully compatible with memory dumps from Windows 10.

Furthermore, there was a discrepancy regarding the use of the *apihooks* command. While both the GitHub version of VolMemLyzer and the paper claimed to use this command, the dataset accompanying the paper did not include features that would be generated by *apihooks*, suggesting that it was likely not used in the original paper.

To get the features in the same format as in the original paper, we modified VolMemLyzer so that it would not execute the *sockets*, *connections*, and *apihooks* commands. This allowed us to get the exact same features as in the dataset provided with the original paper.

We also ran into issues with installing Volatility, due it being written in the now outdated python 2. To overcome those issues, we set up a virtual machine running Ubuntu 16.04. This allowed us to successfully execute Volatility, and therfore also VolMemLyzer, which requires Volability to work.

3.2.2 Using AFF4 files in VolMemLyzer. Another challenge we faced was that Volatility does not support the AFF4 format out of the box. Although there is a plugin intended to enable AFF4 support, it depends on the pyaff4 library, which is currently in a non-functional due to various issues, including incorrect argument usage in its internal methods [5].

This meant that we had to decode the data inside of the AFF4 container into a RAW memory dump that Volability could read.

A number of tools claim to be able to do this. We tested several of these, including winpmem (and its Linux counterpart, linpmem) [4]. However, our attempts at using winpmem were unsuccessful; it failed to produce any output or error messages, simply exiting with the status code 0 on both Windows and Linux.

Similarly, we encountered issues with another tool, aff4imager [2]. Like winpmem, aff4imager also produced no output or errors, so we were not able to use that either.

In addition to these tools, we explored the official AFF4 Java library as a potential solution [7]. Unfortunately, this too proved unsuccessful. Our AFF4 files were missing the 'version.txt' file, which specifies the encoding version of AFF4 used and is required by the library. As a result, the Java library was unable to open our

```
function decode_aff4_file(path) {
    path_no_extension = remove_file_extension(path)
    archive = open_zip_archive(path)

    segment_paths = get_all_archive_entries(archive)
        .filter(e -> e.starts_with("PhysicalMemory/data"))
        .exclude(e -> e.ends_with("index"))

    temp_datafile = create_file(path_no_extension + "-temp")

    for each segment_path in segment_paths
        indexes = read_file_as_int32_array(archive, "$segment_path/index")

        beavy_content = read_raw_data(archive, segment_path)

        for each index in indexes
            next_index = next index in indexes or end of file if index is the last one
            content = zlib_decode_data(bevy_content[index..next_index])
            temp_datafile.write(content)

    yaml_file = read_file_as_yaml(archive, "PhysicalMemory/information.yaml")
    segment_locations = read_yaml_array(yaml_file["Runs"])

    final_ouput_file = create_file(path_no_extension + ".raw")
    file_pointer = 0
    for each segment_location in segment_locations
        data = temp_datafile.read_at(file_pointer, segment_location.length)
        file_pointer += segment_location.length

        final_output_file.write_at(segment_location.start, segment_location.length, data)

    data_file.close()
    data_file.delete()

    final_output_file.close()
}
```

Listing 1: Pseudocode for our AFF4 file decoder

AFF4 files. We were also not able to add version.txt to the archive as the archive contained both folders and files, where the folders had the exact same name and path as the files. This meant that we were unable to correctly extract the AFF4 zip archive, as neither ext4 nor NTFS allow this.

After being unable to use any publicly available tool to decode the raw image from AFF4, we decided to develop our own tool to extract raw data from the AFF4 files. This tool was written in Rust and is specifically designed to work on the files provided with the dataset. A pseudocode showing how this tool decodes an AFF4 file can be seen in listing 1

3.3 Classifier setup

To classify the samples as either benign or malicious, we employed a stacking classifier. The structure of this classifier included base learners—Naive Bayes, Random Forest, and Decision Tree—with Logistic Regression serving as the meta-learner. This configuration mirrors the setup used in the original paper.

The models were all used with their default parameters. The original paper did not go into parameter tuning, which leads us to believe that the default parameters were also used there, so we decided to keep the same setup to be consistent with the original paper.

The implementation was done in Python using the Scikit-learn library [8].

The whole pipeline for classifying samples from the validation dataset is shown in figure 1.

4 RESULTS

4.1 Reproducing the papers results

When using the described two-layer stacked ensemble model with Naive Bayes, Random Forest, and Decision Tree classifiers as the

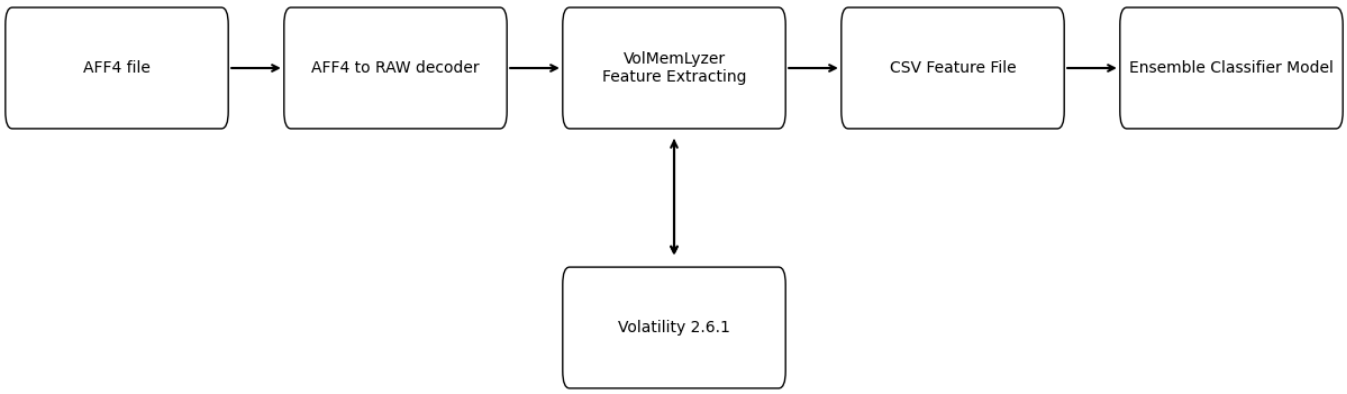

Figure 1: The overview of he classification process.

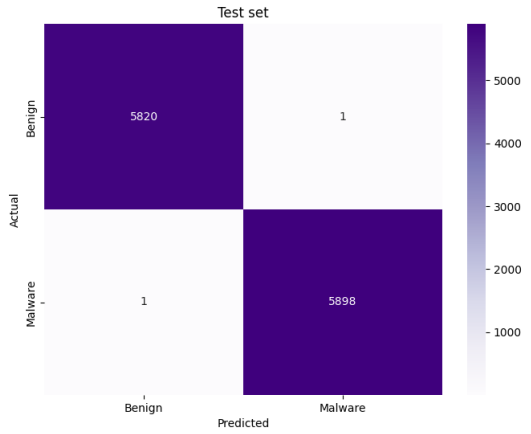

Figure 2: Confusion matrix of the test set classification.

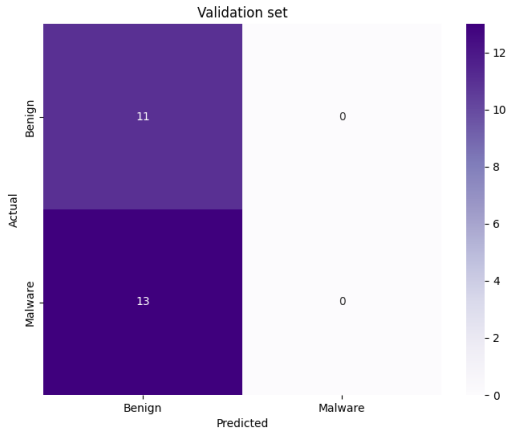

Figure 3: Confusion matrix of the validation set classification.

base learners and Logistic Regression classifier as the meta learner, we successfully managed to reproduce the results of the paper, as can be seen in Table 1 when using the dataset they created using VolMemLyzer on 2,916 malware samples which resulted in 58.596 rows of extracted features from memory samples [1]. The training set consisted of 80% of the set or 46.876 samples, which left the test set with 11720 samples. Only two incorrect classifications were made, as seen in the confusion matrix in Figure 2.

4.2 Validating the results

Using the trained model on the validation set, we did not get any correct malware classifications, as seen in Table 2. Due to time constraints, we only processed 24 memory snapshots from the validation set. 11 benign and 14 malware memory snapshots from the dataset presented in the data article "Memory snapshot dataset of a compromised host with malware using obfuscation evasion techniques" by Sadek, Chong, Rehman, Elovici, and Binder were used [9]. The malware categories were three different types of reverse shells. We processed the memory snapshots with our AFF4 to RAW format decoder and then used the VolMemLyzer to extract features from the RAW output. The classifier predicted all samples' benign, as seen in Figure 3 confusion matrix.

Metric	Original Model	Our Model
Precision	0.99	0.9994
Recall	0.99	0.9997
F1-score	0.99	0.9996
Accuracy	0.99	0.9996

Table 1: Performance Metrics for the original model and our reproduced model

5 DISCUSSION

5.1 Dataset Differences

The differences in the two datasets are apparent. One difference is the different formats, one being a CSV file with 58,596 rows [1], which was processed by the authors of the original paper from RAW format memory snapshots captures, and the other dataset

Metric	Test Set	Validation Set
Accuracy	0.999561	0.458333
Precision	0.999413	0.0
Recall	0.999707	0.0
F1-score	0.999560	0.0
Area under ROC curve	0.999562	0.5

Table 2: Performance Metrics for Test and Validation Sets

being AFF4 formatted memory snapshots [9], which then had to be converted into a format that VolMemLyzer accepted [6]. The other main difference is the malware samples; they are different classes. The original paper's categories are spyware, ransomware, and trojans. In contrast, the categories of the validation dataset were three different types of reverse shells, or more accurately, reverse meterpreter shells, Shellter shells, and Hyperion and PEScrambler shells.

5.2 Issues with the original dataset

They stated in the original paper that the dataset was generated by automating a process where 2,916 malware samples were executed on a VM, and then on a 15-second interval, they extracted the memory snapshot from outside of the virtual machine, and then VolMemLyzer was run on said snapshots to create a dataset in a CSV format where each row represents a memory snapshot [1]. In the dataset shared, there were some inconsistencies. They mentioned they ran 2,916 malware samples, but the outcome in the CSV dataset was 29,298 malware instances.

If they did extract memory snapshots at a 15-second interval ten times for every 2,916 malware samples they claimed to have, then that number, assuming every extraction, was successful, should be 29,160. This is not the case; the actual number in the CSV is 29,298. The column "Category" is either "Benign" or the name of the VM starting which follows the format "{name of the category}{some unique id}-{snapshot number from 1-10}.raw".

It took us only a short time to spot duplicate instances with the same values in every column. This could suggest that the difference between 29,298 and 29,160 is the number of duplicate instances, or 138. However, on further inspection, if we load the dataset into a Pandas data frame, we can see only 28,346 unique values in the Category column, one being "Benign" and the rest following the above format. So, that means we have 28,345 unique malware memory snapshots, which suggest 953 duplicate instances. Figure 4 shows how the naming scheme works and is an example of a duplicate instance.

5.3 Missing feature

When it came to extracting the features out of the memory snapshots, we ran into the issue that VolMemLyzer was configured for Windows 7 images [6], so we had to configure it ourselves by commenting out lines that used modules that were either not mentioned in the original paper.

But one module did not work for us and that was the module "apihooks". In the original paper, they mention the use of the apihooks module [1], but on further inspection, the CSV dataset did follow a naming scheme where every feature had the module name as a prefix. There was no feature with the prefix "apihooks" as can be seen in Table 3.

5.4 Classification results

We do have some severe reservations about the significance of the classification results of the original paper. Even when the duplication issue is factored out, we believe that the nature of the dataset and how it is used to train on that it is always destined to have superb metrics, but only because of how each of the ten memory snapshots taken of each run is so incredibly similar that the model is essentially overfitting greatly.

The results of the validation set were something to be expected due to the differences in the malware categories. However, if you consider that the model was classifying samples that had a near identical or even, in some cases, actually had an identical sample of which the model was trained. Then I think it is fair to say that the model is terrible at generalizing since it was fed, on average, eight very similar snapshots, which we assume because of the ten snapshots on the 15-second interval method they followed and on a VM that may or may not be fresh installs with probably not much or anything running in the background for those 150 seconds. To demonstrate this we trained on 20% of the data and classified 80% of it. The results can be seen in Table 4.

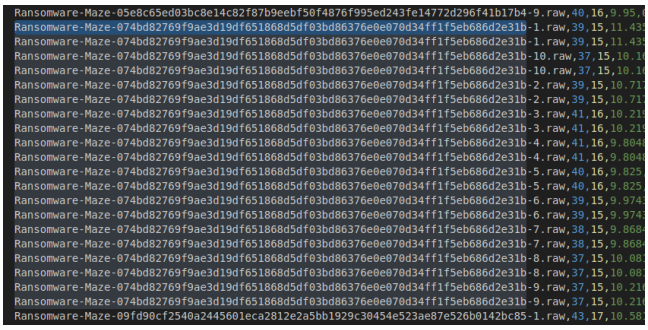

Figure 4: Example of a duplicate instance in the original dataset.

6 CONCLUSION

This research goal was to reproduce and validate the research paper "Detecting Obfuscated Malware using Memory Feature Engineering" by Carrier et al [1]. We ran into several challenges throughout, but in the end, we managed to successfully replicate the original paper using the dataset they provided and their descriptions of the classification model they implemented.

However, the validation aspect of our research did not show that the classifier trained on the malware they trained on could be general enough to classify other types of malware accurately.

This research does have a notable contribution, a tool we developed to convert AFF4 memory images into a RAW format compatible with the widely used tool Volatility. This was one of the critical obstacles of our research in attempting to create a validation set because the only dataset of memory snapshots of malware and benign samples being run on a VM was in the before mentioned AFF4

pslist.nproc	pslist.nppid
pslist.avg_threads	pslist.nprocs64bit
pslist.avg_handlers	dlllist.ndlls
dlllist.avg_dlls_per_proc	handles.nhandles
handles.avg_handles_per_proc	handles.nport
handles.nfile	handles.nevent
handles.ndesktop	handles.nkey
handles.nthread	handles.ndirectory
handles.nsemaphore	handles.ntimer
handles.nsection	handles.nmutant
ldrmodules.not_in_load	ldrmodules.not_in_init
ldrmodules.not_in_mem	ldrmodules.not_in_load_avg
ldrmodules.not_in_init_avg	ldrmodules.not_in_mem_avg
malfind.ninjections	malfind.commitCharge
malfind.protection	malfind.uniqueInjections
psxview.not_in_pslist	psxview.not_in_eprocess_pool
psxview.not_in_ethread_pool	psxview.not_in_pspcid_list
psxview.not_in_csrss_handles	psxview.not_in_session
psxview.not_in_deskthrd	psxview.not_in_pslist_false_avg
psxview.not_in_eprocess_pool_false_avg	psxview.not_in_ethread_pool_false_avg
psxview.not_in_pspcid_list_false_avg	psxview.not_in_csrss_handles_false_avg
psxview.not_in_session_false_avg	psxview.not_in_deskthrd_false_avg
modules.nmodules	svcscan.nservices
svcscan.kernel_drivers	svcscan.fs_drivers
svcscan.process_services	svcscan.shared_process_services
svcscan.interactive_process_services	svcscan.nactive
callbacks.ncallbacks	callbacks.nanonymous
callbacks.ngeneric	

Table 3: Table showing all the features in the dataset from the original paper.

Metric	20% train - 80% test	80% train - 20% test
Precision	0.9994	0.9994
Recall	0.9997	0.9997
F1-score	0.9996	0.9996
Accuracy	0.9996	0.9996

Table 4: Performance Metrics for the comparison of a 80/20 and 20/80 training/test split.

format and was not compatible with VolMemLyzer which utilizes before-mentioned Volatility to extract the features from the memory images [6]. The AFF4 to RAW format decoder is available on the public GitHub repository: https://github.com/gudmunduro/mlcs-final.

In conclusion, while our study reaffirms parts of the original paper, we highlight its limitations and flaws. Future work should focus on enhancing the model to be more adaptable and diverse when classifying samples different from the samples the model was trained on. For instance, refrain from contaminating the training and test sets by having any of the ten snapshots of the same VM execution group reside in both the test and training sets.

REFERENCES

[1] Tristan Carrier, Princy Victor, Ali Tekeoglu, and Arash Habibi Lashkari. 2022. Detecting Obfuscated Malware using Memory Feature Engineering. In *Proceedings of the 8th International Conference on Information Systems Security and Privacy (ICISSP 2022)*. Canadian Institute for Cybersecurity (CIC), University of New Brunswick (UNB), Fredericton, NB, Canada; Johns Hopkins University Applied Physics Laboratory, Critical Infrastructure Protection Group, Maryland, U.S.A., 177–188.

[2] Michael Cohen. 2018. *AFF4 Documentation.* https://docs.aff4.org/en/latest/

[3] Michael Cohen, Simson Garfinkel, and Bradley Schatz. 2009. Extending the advanced forensic format to accommodate multiple data sources, logical evidence, arbitrary information and forensic workflow. *digital investigation* 6 (2009), S57–S68.

[4] Velocidex Developers. 2023. *Velocidex c-aff4 Repository.* https://github.com/Velocidex/c-aff4

[5] Volatility Foundation. 2019. *Volatility Community Repository.* https://github.com/volatilityfoundation/community

[6] Ali Hadi Lashkari. 2023. VolMemLyzer: A Tool for Volatile Memory Analysis. https://github.com/ahlashkari/VolMemLyzer. Accessed: 2023-11-25.

[7] Schatz Forensic Pty Ltd. 2019. *AFF4 Java Repository.* https://github.com/aff4/aff4-java

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[9] I. Sadek, P. Chong, S. U. Rehman, Y. Elovici, and A. Binder. 2019. Memory snapshot dataset of a compromised host with malware using obfuscation evasion techniques. *Data in brief* 26 (2019), 104437. https://doi.org/10.1016/j.dib.2019.104437

[10] Volatility Foundation. 2020. Volatility: An advanced memory forensics framework. https://github.com/volatilityfoundation/volatility. Accessed: 2023-11-25.