CHALMERS
UNIVERSITY OF TECHNOLOGY

All Malware Looks Similar Until It Doesn't

Learning Malware Embeddings for Similarity Search and Zero-Shot Generalization Across Static, Dynamic, and Hybrid Representations

Master's thesis in Complex Adaptive Systems

GUSTAF FRANSÉN BJÖRN & ROBIN OSCARSSON

DEPARTMENT OF PHYSICS

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Master's thesis 2026

All Malware Looks Similar Until It Doesn't

Learning Malware Embeddings for Similarity Search and Zero-Shot Generalization Across Static, Dynamic, and Hybrid Representations

Gustaf Fransén Björn
Robin Oscarsson

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2026

All Malware Looks Similar Until It Doesn't
Learning Malware Embeddings for Similarity Search and Zero-Shot Generalization Across Static, Dynamic, and Hybrid Representations
Gustaf Fransén Björn
Robin Oscarsson

© Gustaf Fransén Björn & Robin Oscarsson, 2026.

Supervisor: Anders Hansson, Recorded Future
Examiner: Mats Granath, Department of Physics

Master's Thesis 2026
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Malware family grouped cosine similarity in the scaled EMBER feature space, where each family is represented by 50 samples.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2026

All Malware Looks Similar Until It Doesn't
Learning Malware Embeddings for Similarity Search and Zero-Shot Generalization Across Static, Dynamic, and Hybrid Representations
Gustaf Fransén Björn
Robin Oscarsson
Department of Physics
Chalmers University of Technology

Abstract

This thesis investigates the use of machine learning to learn embeddings from static, dynamic, and hybrid malware representations. The learned embeddings are evaluated through similarity search and zero-shot clustering, with particular focus on their ability to generalize beyond malware families observed during training. To this end, Siamese metric learning approaches based on triplet loss are applied using multi-layer perceptrons for static, dynamic, and hybrid representations, and a transformer-based encoder for dynamic malware reports. The results highlight the inherently difficult nature of learning semantically meaningful malware embeddings that generalize to unseen malware families. In particular, the thesis shows that strong performance on unseen samples from seen families in training does not necessarily imply true generalization. Overall, dynamic and hybrid representations provide a stronger basis for similarity learning than only static representations. The findings further suggest that the main limitation is not necessarily the model architecture, but mostly the noisy, imbalanced, and ambiguous nature of malware data itself.

Keywords: malware analysis, malware embeddings, metric learning, Siamese networks, similarity search, zero-shot generalization, hybrid malware representations, transformer encoders, EMBER, dynamic malware analysis

Acknowledgements

We would like to express our sincere gratitude to Recorded Future, and especially the Malware Intelligence team, for giving us the opportunity to conduct this thesis in such a supportive and encouraging environment. We would especially like to thank Anders Hansson, our supervisor at Recorded Future. Without his insightful discussions, continuous support, and guidance throughout the project, this work would have been significantly more challenging.

We also want to thank our academic supervisor and examiner, Mats Granath, for his valuable support and feedback during the thesis project.

In addition, we extend our appreciation to the previous contributors August Klynne and Malte Åqvist, whose thesis project in 2025 at Recorded Future started to explore the area on which this thesis is based. Their contributions were highly inspiring and helped shape the direction of this work.

Finally, we would like to thank Chalmers University of Technology and the Department of Physics for providing an enriching and challenging educational environment during the past five years. The knowledge and experience gained during our studies were instrumental in completing this thesis.

Gustaf Fransén Björn
Robin Oscarsson
Gothenburg, May 2026

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AI	Artificial Intelligence
API	Application Programming Interface
AV	Antivirus
BERT	Bidirectional Encoder Representations from Transformers
BM25	Best Matching 25
DNS	Domain Name System
ELF	Executable and Linkable Format
EMBER	Endgame Malware BEnchmark for Research
GELU	Gaussian Error Linear Unit
HTTP	Hypertext Transfer Protocol
IDF	Inverse Document Frequency
kNN	k-Nearest Neighbors
ML	Machine Learning
MLM	Masked Language Modeling
MLP	Multi-Layer Perceptron
PE	Portable Executable
PK	P classes with K samples per class batching
t-SNE	t-distributed Stochastic Neighbor Embedding
YARA	Yet Another Recursive Acronym

Nomenclature

Below is the nomenclature of the most important symbols and variables used throughout this thesis.

Sets and Datasets

B	Training batch
$\mathcal{D}$	Embedding database
$\mathcal{P}_a$	Set of valid positive samples for anchor a
$\mathcal{N}_a$	Set of valid negative samples for anchor a

Embeddings and Metric Learning

s_i	Input sample i
y_i	Class label for sample i
e_i	Embedding vector for sample i
z_i	Normalized embedding vector for sample i
f_θ	Siamese embedding model parameterized by θ
d_{emb}	Embedding dimensionality
D_{ij}	Pairwise distance between embeddings i and j
D_{ap}	Distance between anchor and positive sample
D_{an}	Distance between anchor and negative sample
θ_{ij}	Angle between normalized embeddings
a	Anchor sample in triplet loss
p	Positive sample in triplet loss
n	Negative sample in triplet loss
m	Triplet-loss margin
L_{batch}	Batch triplet loss
P	Number of classes sampled in PK batching

K	Number of samples per class in PK batching

Transformer Variables

M	The set of selected masked token positions in MLM
H_i	Token-level hidden representations for sequence i
h_{ij}	Hidden representation for token j in sequence i
S_i	Sentence/report-level embedding
n_i	Number of tokens in sequence i

Retrieval and Similarity

q	Query embedding or query document
$\mathrm{sim}(q, x)$	Similarity function between query q and sample x
$N_k(q)$	Set of k nearest neighbors for query q
$f(t, d)$	Term frequency of term t in document d
$\mathrm{IDF}(t)$	Inverse document frequency for term t
$\lvert d \rvert$	Document length
avgdl	Average document length in the corpus
k_1	BM25 term frequency scaling parameter
b	BM25 length normalization parameter

Contents

List of Acronyms **ix**

Nomenclature **xi**

List of Figures **xvii**

List of Tables **xxi**

1 Introduction **1**
1.1 Aim . . . 2
1.2 Research Questions . . . 2
1.3 Limitations . . . 3
1.4 Thesis Disposition . . . 3
1.5 The Use of AI . . . 3

2 Background **5**
2.1 Malware Analysis . . . 5
2.1.1 Static vs. Dynamic Analysis . . . 6
2.1.2 Machine Learning in Malware Analysis . . . 6
2.2 Industry Applications . . . 7
2.2.1 Recorded Future . . . 8

3 Related Work **9**
3.1 Previous Research on Static Analysis . . . 9
3.2 Previous Research on Dynamic Analysis . . . 9
3.3 Previous Research on Hybrid Analysis . . . 10
3.4 Previous Research on Metric Learning . . . 10
3.5 Similarity-Based Malware Analysis . . . 11
3.6 Research Gap . . . 11

4 Theory **13**
4.1 Static Malware Analysis . . . 13
4.1.1 PE Files . . . 13
4.1.2 EMBER Features . . . 14
4.2 Dynamic Malware Analysis . . . 15
4.2.1 Dynamic Reports . . . 15
4.3 Metric Learning . . . 16

4.3.1 Siamese Networks 16
4.3.2 Triplet Loss 16
4.3.3 *PK* Batch Sampling 19
4.4 Transformers Applied to Metric Learning 20
4.4.1 The Basis of Transformers 20
4.4.2 BERT 22
4.4.3 Sentence Transformer 24
4.4.3.1 Report Pooling 25
4.5 Retrieval and Reranking 25
4.5.1 k-Nearest Neighbors 25
4.5.2 Best Matching 25 26
4.6 Evaluation Metrics 26
4.6.1 Davies-Bouldin Index 26
4.6.2 t-SNE 27
4.6.3 Cosine Similarity 28

5 Methods 29
5.1 Data 29
5.1.1 Static Data 30
5.1.1.1 Manual Feature Extraction 30
5.1.1.2 Preprocessing 31
5.1.2 Dynamic Data 31
5.1.2.1 Manual Feature Extraction 32
5.1.2.2 Preprocessing 33
5.1.3 Hybrid Data 34
5.1.4 Preprocessing of Dynamic Reports 34
5.1.4.1 Transformer Encoders 34
5.1.4.2 BM25 36
5.1.5 Dataset Splits 36
5.1.5.1 Deduplication 37
5.1.5.2 Transformer 39
5.2 Embedding Models 39
5.2.1 Siamese MLP 40
5.2.1.1 Training 40
5.2.2 BM25 Reranking 42
5.2.3 Malware Adapted Siamese ModernBERT 42
5.2.3.1 MLM Pre-Training 43
5.2.3.2 Siamese Fine-Tuning 43
5.3 Evaluation 45

6 Results 47
6.1 Siamese MLPs 47
6.1.1 Single Modality Embeddings 47
6.1.2 Hybrid Embeddings 48
6.1.3 Deduplication and Reranking 50
6.2 Malware Adapted Siamese ModernBERT 52
6.2.1 Training and Validation for MLM Pre-training 52

6.2.2 Training and Validation for Siamese Fine-Tuning 53
6.2.3 Evaluation of Embeddings 53

7 Discussion 57
7.1 Static vs. Dynamic Malware Representations 57
7.2 Hybrid Representations and Learning 59
7.3 Transformers and Their Ability to Generalize 59
7.4 Data Quality and Malware Family Ambiguity 60
7.5 Retrieval Refinement with BM25 Reranking 62
7.6 The Challenge of Evaluating Zero-Shot Generalization on Malware . . 62
7.7 Operational Implications 63
7.8 Ethical Considerations 63

8 Conclusion 65

9 Future Work 67

Bibliography 69

A EMBER features I

B Manually Extracted Dynamic Features III

C Dynamic Report Sections XIII

D Additional Results XVII

List of Figures

4.1 Illustration of the triplet-loss learning objective. During training, the embedding model learns to move positive samples closer to the anchor while simultaneously increasing the distance to negative samples in the embedding space. 16
4.2 Illustration of triplet-loss sampling. Hard negatives lie closer to the anchor than the positive sample, semi-hard negatives lie outside the positive distance but within the margin, and easy negatives lie outside the margin boundary. 17
4.3 A picture of the original transformer architecture. Image taken from the original "Attention Is All You Need" paper [1]. 21
4.4 This figure shows conceptually how an input sequence of tokens is encoded through BERT. It takes the sequence, adds certain special tokens such as [CLS] and [SEP] to signify start and stop of a sequence, and finally creates embedding vectors for all tokens in a sequence. The [CLS] token is typically used to represent the full sequence. Image by Daniel Voigt Godoy, licensed under CC BY 4.0 [2]. 22
4.5 This figure shows conceptually how the BERT model is trained using MLM. A certain amount of tokens are masked, as shown by the [Mask] symbol, and the objective is for the model to predict which word was there originally based on the bidirectional context. Notice how the masked token is fed through BERT to produce a hidden representation, then through a feed-forward network to produce logits. Image by Daniel Voigt Godoy, licensed under CC BY 4.0 [3]. 23

5.1 Overview of the malware embedding pipeline. Everything starts from the malware samples, that are analyzed both statically and dynamically. The malware is then processed, and fed to machine learning models used to create the embeddings. Both static and dynamic embeddings were created and analyzed. Hybrid embeddings were created both by feeding a model static and dynamic features, as well as fusing embeddings from static and dynamic models, and then analyzing them. In the case of dynamic models, both Siamese MLP and transformers are used to create embeddings. Note that the Siamese transformer in this case refers to BERT transformer encoders. A BM25 reranking method on the hybrid embeddings was also implemented and analyzed. 29

5.2 Distribution of inter- and intra-family similarity of EMBER features after the scaler was applied. Before scaling, the distributions were just one small peak around zero, and a much larger one close to one. But by preprocessing the data as described above, differences are magnified and a more distributed cosine similarity can be observed, although one can still see a very large peak close to one, indicating that the samples in families do not differ much, leaving very little room for a model to learn. 38
5.3 The figure shows the fraction of samples kept per malware family after deduplication. The large variation between families indicates that duplicate density differs across the dataset, suggesting that some malware families consist primarily of near-duplicates while others are more diverse. 38
5.4 Conceptual overview of the domain-adaptation training approach used in this thesis. First, ModernBERT-base is pre-trained on dynamic malware reports using MLM. The resulting encoder is then fine-tuned in a Siamese metric learning setup using triplet loss. Although the figures refer generally to BERT-style encoders, the model used in this thesis is ModernBERT-base. 44

6.1 t-SNE visualizations of the hybrid representation before and after Siamese training, for both the test set and the set of unseen samples from families seen during training. This is only shown for the hybrid model, not the fusion model, that also showed the same pattern. Notice the well separated clusters in the (b)-plot, those are the unseen samples from families seen during training. The unseen families struggle to form meaningful separated clusters. This shows the difficult task of inter-family generalization, compared to intra-family generalization. 49
6.2 t-SNE visualizations of the hybrid embedding space on the deduplicated dataset before and after Siamese training, shown for both the hold-out test set and unseen samples from families observed during training. While the model still clusters seen families reasonably well, the deduplicated setting produces more overlap than the non-deduplicated results in Figure 6.1. In contrast, unseen families fail to form clearly separated clusters, highlighting the difficulty of inter-family zero-shot generalization compared to intra-family generalization. 51
6.3 Training and evaluation loss curves for the MLM pre-training stage of ModernBERT. Note that the training loss (solid line) is logged throughout training, while the evaluation loss (dashed line) is computed less frequently on the zero-shot validation set. 52
6.4 Training (solid line) and evaluation (dashed line) loss curves for the Siamese metric learning fine-tuning stage of the domain-adapted ModernBERT model. Note that the training loss is logged throughout training, while the evaluation loss is computed less frequently on the zero-shot validation set. 53

6.5 *t*-SNE visualisations of report-level embeddings produced by the original ModernBERT model as reference, using a sequence length of 1026 for each chunk. Each subplot shows 3000 samples from one dataset split: the training set, the internal validation set, the zero-shot validation set, and the hold-out set. 55
6.6 *t*-SNE visualizations of report-level embeddings produced by the domain-adapted ModernBERT model using a sequence length of 1026. Each subplot shows 3000 samples from one dataset split: the training set, the internal validation set, the zero-shot validation set, and the hold-out set. 56

7.1 Distribution of inter- and intra-family cosine similarity of manually extracted dynamic feature vectors after preprocessing and scaling. Compared to the static EMBER representation in Figure 5.2, the dynamic representation exhibits broader intra-family variation and greater overlap between malware families. 58
7.2 Family grouped cosine similarity in the scaled EMBER feature space, where each family is represented by 50 samples. Although some malware families form local similarity clusters, many families remain close to orthogonal in the feature space, indicating sparse statistical separation rather than globally shared semantic structure, aligning with Figure 5.2. 61

D.1 t-SNE visualizations of the open-set evaluation space for static and dynamic representations before and after Siamese training. Notice the well separated clusters in the trained files, those are the unseen families from families seen during training. The unseen families struggle to form meaningful separated clusters. XVII

List of Tables

4.1 Overview of the EMBER feature categories and the information they encode. The extracted raw features are vectorized before being used as input to machine learning models [4]. 14

4.2 Dynamic report sections used in this thesis. 15

5.1 Selected EMBER feature groups and preprocessing pipeline used in this work. Excluded features include categorical header fields, detailed section attributes, import/export features, and other high-cardinality sparse metadata. 32

5.2 Architecture overview of the final selected Siamese MLP embedding models evaluated in this work. All models used batch normalization and dropout regularization between hidden layers. The activation function used was GELU, and the dropout rate was set to 0.2. The notation under MLP architecture describes the input dimension, hidden layer dimension, and lastly embedding dimension. 40

5.3 Training configuration and hyperparameter selection used for the Siamese MLP models. The deduplicated hybrid model used PK-batching due to a limited amount of samples from some families, using $P = 32, K = 16$. Fusion architectures were evaluated during tuning, although the final fusion representation used for evaluation was direct concatenation. 41

5.4 Training configurations for MLM pre-training and Siamese metric learning fine-tuning. 43

6.1 Retrieval performance for static and dynamic embedding models before and after Siamese training. Purity@10 measures the proportion of retrieved neighbors belonging to the same malware family as the query sample, while Hit@10 measures whether at least one retrieved neighbor belongs to the same family. Purity is a per sample average, while hit rate is a per family average, to get both perspectives. The raw evaluation is the feature space, a much higher dimensional space than the embedding space. Since the metrics are dataset dependent, numerical comparisons between the unseen and seen family test cannot be made. 48

6.2 Retrieval performance for hybrid embedding models before and after Siamese training. Purity@10 measures the proportion of retrieved neighbors belonging to the same malware family as the query sample, while Hit@10 measures whether at least one retrieved neighbor belongs to the same family. Purity is a per sample average, while hit rate is a per family average, to get both perspectives. The raw evaluation is the hybrid feature space, a much higher dimensional space than the embedding space. Since the metrics are dataset dependent, numerical comparisons between the unseen and seen family test cannot be made. 50
6.3 Retrieval performance for the deduplicated hybrid embedding models before and after Siamese training, and with the added BM25-reranking of the 100 nearest neighbors from the embedding space. Purity@10 measures the proportion of retrieved neighbors belonging to the same malware family as the query sample, while Hit@10 measures whether at least one retrieved neighbor belongs to the same family. Purity is a per sample average, while hit rate is a per family average, to get both perspectives. The raw evaluation is the hybrid feature space, a much higher dimensional space than the embedding space. Since the metrics are dataset dependent, numerical comparisons between the unseen and seen family test cannot be made. 52
6.4 Evaluation metrics for Purity@10, Davies-Bouldin index, and Hit@10 for the baseline ModernBERT model and the domain adapted ModernBERT model across the four dataset splits. Purity@10 measures the proportion of retrieved neighbors belonging to the same malware family as the query sample, while Hit@10 measures whether at least one retrieved neighbor belongs to the same family. Purity is a per sample average, while hit rate is a per family average, to get both perspectives. Note that hit-rate is only reported for the hold-out set, since the metric is most meaningful for families with few samples. For families with more samples, the interesting metric is purity. 54

B.1 Overview of extracted network features, including their description and motivation. Many of these features are inspired by prior work, for example [5]. Note that one report corresponds to one malware sample. III
B.2 Signature features extracted from the dynamic reports. IX
B.3 Process features extracted from the dynamic reports. XI
B.4 Dumped file features extracted from the dynamic reports. XII

1
Introduction

In today's volatile world, geopolitics and artificial intelligence (AI) have become defining forces in cybersecurity. Cyber operations are now a central component of modern conflicts, where both state-sponsored and criminal threat actors use increasingly sophisticated methods to target critical infrastructure, organizations, and individuals. Simultaneously, advances in artificial intelligence are transforming both offensive and defensive cybersecurity capabilities. AI enables automated threat detection and analysis on a large scale for defenders, while at the same time allowing attackers to execute more scalable and adaptive cyberattacks [6].

The impact of AI on cybersecurity is complex. Which side that will gain most from this remains unclear, but it is clear that AI is pushing cybersecurity into new and unexplored territory [7]. Today, a large part of the cybersecurity domain consists of analyzing malware, which is classified by different family labels. These labels work very well for known or structurally similar samples to what is known today, as they indicate the function of the malware and the threat actor associated with it; giving insights into how and why an attack is taking place. However, some malware samples cannot be automatically attributed to a known malware family, either due to malware families evolving over time or due to the emergence of completely novel families, and they require extensive manual analysis. This is typically done by identifying groups of similar files and extracting generic patterns shared among them, a process that is time-consuming and requires deep expertise. A system that enables analysts to query directly for similar samples would therefore greatly accelerate and democratize malware analysis, giving insights before the time-consuming manual analysis is finished. One promising approach is to represent malware samples as vector embeddings and store them in a vector database, enabling efficient similarity search and pattern discovery.

In light of that, Recorded Future had a separate master thesis during 2025 where they started exploring vector embeddings of malware [8]. In that thesis, they explored the embeddings of both dynamic and static malware representations by embedding the former with Best Matching 25 (BM25) and transformer methods, and the latter by using EMBER (Endgame Malware BEnchmark for Research) features, an open source method to extract features from portable executable (PE) files also serving as a benchmark dataset for researchers [4]. Their primary goal was to embed dynamic reports to find richer, behavior based clusters and labels than the family labels attached to malware samples, as these labels differ between vendors and have a tendency to be created in an ad-hoc manner [9]. They then tried to

embed the static EMBER features using a Siamese neural network to map samples to the formed behavior based clusters, in order to extract a richer representation without the need of the more computationally expensive sandbox runs needed to generate dynamic reports. Their findings indicated that static EMBER features contain insufficient behavioral information to reproduce the structure captured by dynamic representations.

However, this result also raises a more fundamental question. Before asking how static and dynamic malware representations map to each other, it is necessary to ask whether either representation can form a basis for semantically meaningful embedding spaces in the first place. Here, a semantically meaningful embedding space is understood as one in which malware samples are close because they share relevant behavioral properties. In particular, it is unclear whether such embeddings can form clusters that reflect malware behavior and generalize to samples from previously unseen families. This thesis therefore focuses on the more general problem of whether static and dynamic malware representations can be embedded in a way that supports meaningful similarity search and zero-shot clustering.

Particular attention is given to hybrid embeddings, utilizing both static and dynamic representations, with the hypothesis that hybrid embeddings would outperform single-modality embeddings for similarity search, generalization, and (zero-shot) clustering. Research into a different use case, malware classification, using a hybrid approach shows improved accuracy in this task [10, 11], but similarity and clustering remain less explored areas of research. A semantically meaningful embedding space of malware would enable malware analysts to find similar samples to new and unknown malicious files, improving robustness to better handle the problem of distribution shifts in malware analysis.

1.1 Aim

The aim of this thesis is to leverage static and dynamic representations of malware samples and train Siamese neural networks on these representations, in order to create semantically meaningful embeddings of malware. Ultimately, the goal is to develop a method to efficiently cluster and retrieve similar malware samples using this embedding space, while generalizing to unseen and modified malware families through zero-shot similarity and clustering.

1.2 Research Questions

The project addresses the following research questions:

1. How can static, dynamic, and hybrid malware representations be used to create semantically meaningful embedding spaces for malware similarity search and clustering?

2. To what extent can these embeddings and particularly hybrid embeddings trained on frequent malware families, generalize to unseen or rare families (zero-shot generalization)?

This was investigated through metric learning approaches using Siamese multi-layer perceptrons (MLPs) and transformer encoder architectures, combined with retrieval evaluation, clustering analysis, and zero-shot validation on unseen malware families.

1.3 Limitations

The scope of this work will be limited to:

- Using data available from Recorded Future's internal sandbox (Triage) environment, and EMBER features are used as the static representation.

- Analyzing Windows PE files only; other formats such as ELF, Mach-O, or mobile binaries like APK are excluded.

- Focusing on Siamese neural network architectures for embedding learning and not exploring alternative metric learning approaches extensively.

- Evaluating embeddings primarily through clustering quality, retrieval accuracy, and qualitative visualization (e.g. t-SNE plots).

- Detection rule generation (e.g., Sigma or YARA rules) and large-scale deployment within production systems are outside the scope of this project, though the results could form a foundation for such future work.

1.4 Thesis Disposition

This thesis is structured as follows. Chapter 2 introduces the background of malware analysis and its industrial context. Chapter 3 reviews related work on static, dynamic, hybrid, and similarity-based malware analysis. Chapter 4 presents the theoretical concepts used in the thesis, including metric learning, Siamese networks, transformers, retrieval methods, and evaluation metrics. Chapter 5 describes the data, preprocessing, models, and experimental methodology. Chapter 6 presents the results, while Chapter 7 discusses the findings and their implications. Finally, Chapters 8 and 9 present the conclusion and future work.

1.5 The Use of AI

During this thesis, AI tools, including ChatGPT and Claude, were used as supporting tools for debugging code and proofreading text. The AI tools were not used to generate the thesis content or produce the code used in the project.

2

Background

This chapter introduces the main area of this thesis, namely malware analysis. Malicious actors use malware, **Mal**icious Soft**ware**, to target companies, organizations, and individuals in order to obtain sensitive data, cause chaos, or for financial extortion. To hinder these attacks, billions of dollars are being spent to protect cyber infrastructure, and an important part of this is the analysis of malware.

2.1 Malware Analysis

In 2024, Microsoft reported that their customers were targeted in 600 million identity cyberattacks every day [12]. In order to defend systems against the growing number of attacks, cybersecurity companies have to continuously analyze the malware used, in an attempt to prevent malicious actors from successfully infecting systems. These analyses are performed in various ways, and since the cybersecurity companies want to protect their methods both from competitors and malicious actors, many of these methods are not public. Cybersecurity is, however, also an area in academia, where new papers are published daily, in the everlasting war between attackers and defenders.

The goal of a malware analysis can differ, it is often focused on detection. That is, to distinguish malicious software from benign software so that malware can be detected and prevented from running. But that is not always enough, as sometimes it is necessary to get more insights into how and why a system is being targeted. This is where classification of malware becomes important, and is also a well researched area, where malware is studied on a deeper level in order to classify its function and origin. Malware can be categorized at different levels of granularity, ranging from broad categories such as ransomware or trojans to more specific malware families. Family in this case is a categorization of malware, where malware with similar attack patterns are clustered together [10]. A family can indicate what threat actor is behind it, and what the goal of the attack is, giving defense systems a warning so that they can react in time if the malware is classified correctly.

But, of course, malware actors have knowledge of these defensive methods, and are always modifying and/or obfuscating the malware to avoid detection and classification [13]. It is truly a cat-and-mouse game, with attackers and defenders competing, where attackers always have the great advantage of acting first, forcing defense systems to react to changes. This is a great challenge for the field of cybersecurity,

specifically in finding, detecting, and classifying novel or modified malware before it has already infected a system.

2.1.1 Static vs. Dynamic Analysis

Malware analysis can be divided into separate sections. Firstly static analysis, where malware is analyzed without execution, and secondly dynamic analysis, where the behavior of malware is studied during execution. A third field of malware analysis is the combination of these, hybrid analysis, where both representations are analyzed together in order to obtain information about malware.

Static malware analysis is cheaper than dynamic analysis, since no isolated environment where the malware is executed, often called a "sandbox", is needed. Instead the binary code of the malware is analyzed. This code is not human readable, as it is only meant for a machine to execute. For a human to understand it, reverse engineering is needed, which is a difficult and time consuming process that few people can do. Instead, often statistical features like byte counts, strings, entropy etc. are extracted from the files, in order to get indications of what the file is doing.

Dynamic malware analysis however, is performed by executing the malware and studying its behavior. It is a richer representation of the malware, but is more computationally expensive. The same malware sample can also behave differently between runs, and can detect that it is being run in a sandbox and hide its malicious behavior. So it is not as deterministic as studying the binary code, but can instead give a greater behavioral understanding of the malware.

When combining these approaches in the hybrid malware analysis, the idea is that more information can be extracted and more insights can be found. There are several ways to combine these sources of information. Hybrid analysis attempts to combine the complementary strengths of static and dynamic representations in order to obtain richer malware representations.

2.1.2 Machine Learning in Malware Analysis

A challenge when applying machine learning (ML) methods to malware analysis is feature extraction [14], i.e. determining how malware samples should be represented for learning algorithms. Both static and dynamic malware analysis use a wide variety of representations, ranging from manually engineered statistical features to behavioral reports and sequential execution traces. Consequently, malware machine learning studies employ a broad range of methods that are often difficult to compare due to differences in datasets, preprocessing pipelines, and evaluation methodologies [15].

The objectives of malware analysis also differ between studies. Some focus on malware detection, while others investigate malware classification or similarity learning

between samples. A common challenge across these approaches is dataset quality and evaluation methodology. Many studies rely on private or small datasets, or unclear train and test splits, making results difficult to reproduce and compare [15]. EMBER was introduced as a partial solution to this problem by providing a large public dataset and standardized feature extractor for static malware analysis [4].

Machine learning applied to malware analysis is an inherently difficult problem, especially when focusing on zero-shot learning. Malicious actors try their best to modify their code and avoid detection. They use packers, encryption, and obfuscation etc. to trick the detection systems [16]. A packer for instance compresses the binary file and in some cases encrypts it. If static analysis is run on that file, it will only see the compressed binary and struggle, even if it has the same function as a previously seen sample. Dynamic analysis solves this in some ways, but has other problems, like stochastic execution behavior, meaning a sample does not have to act the same between two runs or sometimes does not run at all in a sandbox.

These challenges make generalization difficult for malware machine learning models, particularly in zero-shot settings involving previously unseen malware families. Models may overfit to artifacts such as packer signatures or family specific statistical fingerprints rather than learning transferable behavioral structure. Furthermore, obfuscation and minor modifications can produce highly similar malware variants that still appear as new samples, making data leakage and evaluation difficult [16]. As a result, malware similarity research often suffers from limited standardization and unclear evaluation methodologies [15]. EMBERsim was introduced in 2024 as a partial solution to standardized similarity-based malware research by extending EMBER with malware family labels and similarity metadata [17], but has not been widely utilized.

2.2 Industry Applications

In the cybersecurity industry, the problems described are even more difficult. Firstly, they work with a completely different scale than in academia, for example Microsoft reports processing more than 78 trillion security signals daily [12], and they need to detect novel and modified samples before they cause harm. The scale makes automated analysis the only solution to the problem, whereas security experts mostly deal with ongoing threats.

Antivirus (AV) programs traditionally use signature based detection and classification. That is, hashing the bytes in the header of the suspicious file, and matching it against seen malware in a database, looking for common signatures that indicate malicious behavior [18]. This method works well for previously seen malware samples, but struggles with new or modified samples since then the header of the file is changed. These operational challenges motivate the need for scalable similarity-based malware representations capable of generalizing to previously unseen samples and malware families.

2.2.1 Recorded Future

Recorded Future, the company this thesis is done in collaboration with, is the world's largest threat intelligence company. The company collects information across all of the internet, to detect and prevent cyberthreats for their customers. They do this by applying natural language processing and ML methods on large amounts of data, and also offer a sandbox for malware analysis. There, the submitted samples are given a risk score and a family label if possible. However, samples that cannot be automatically attributed to a known malware family is left unlabeled and requires deeper analysis for attribution. A system that enables analysts to query directly for similar samples would therefore greatly accelerate and democratize malware analysis. One promising approach is to represent malware samples as vector embeddings and store them in a vector database, enabling efficient similarity search and pattern discovery, thus Recorded Future's interest in this thesis.

3

Related Work

Malware is a well researched area, and in this chapter, a summary of related work will be presented, as well as the relevance of this thesis.

3.1 Previous Research on Static Analysis

Static malware analysis has evolved over time. Early malware analysis methods primarily focused on signatures and expert analysis [19]. These methods were slow and expensive and, although very accurate, not feasible as more and more malware started to circulate. Consequently, focus instead shifted to more automated analysis, based on data mining, feature extraction, signatures, and machine learning models [20, 21].

Fast forward to 2018 and a major development when the EMBER dataset was introduced, a standardized feature extraction framework for PE files and a benchmark dataset for static malware analysis, together with a LightGBM baseline model for malware detection [4]. EMBER outperformed contemporary deep learning methods for detection, like MalConv, that showed that convolutional neural networks on raw byte sequences could be used successfully for malware analysis [22]. Another widely used raw input method is converting malware binaries to gray scale images for deep learning image analysis [23]. Control flow graphs, opcodes, and other structural features are also used.

Despite these advances, static methods often struggle with the fundamental problem of static analysis, namely that binaries do not fully capture runtime behavior [8]. As a result, new, modified, and obfuscated malware is difficult to detect, limiting the generalization capabilities [18], although some research specifically focuses on this task [24, 25].

3.2 Previous Research on Dynamic Analysis

Due to the relatively expensive computational resources needed to produce dynamic data, research on dynamic malware analysis is less prominent than research on static analysis. This is especially the case for deep neural network based methods, which depend on a large amount of data. Moreover, dynamic representations often contain large amounts of information, making it difficult to determine which features and information should be extracted. This remains a challenge even with domain

expertise, since several choices are often available [26].

Earlier research in this area has explored several ways of representing dynamic malware behavior. Some approaches manually extract semantically meaningful dynamic features and apply unsupervised clustering algorithms to group samples with similar behavior [27]. Others focus specifically on application programming interface (API) call-based representations [28], or aim to create richer behavioral representations by tracking information flow during execution [29].

However, due to the complexity and domain-expertise required to parse dynamic malware information, more recent research has started to focus on how transformer architectures with their powerful attention mechanism can be leveraged for malware analysis generally, and dynamic analysis specifically [30]. These approaches range from applying off-the-shelf solutions on the malware domain [8], to developing and fine-tuning domain-specific transformer architectures [31, 32]. However, much of this research, like in static analysis, is focused on how these types of deep learning architectures can be used on malware detection and classification, rather than on retrieval, similarity search, and behavioral clustering.

3.3 Previous Research on Hybrid Analysis

There is previous research that applies machine learning and multimodal learning techniques on hybrid malware analysis [10, 11, 33, 34, 35, 36]. However, differences in datasets, feature extraction methods, general methodology and evaluation protocols make direct comparisons between studies difficult. Moreover, most of the research focuses on either detection or classification as well. Meaning that relatively little research exists on similarity learning, embedding quality, and generalization to unseen malware families, leaving a gap in the development of standardized methods for zero-shot hybrid malware analysis.

3.4 Previous Research on Metric Learning

Metric learning aims to learn embedding spaces where semantically similar samples cluster together, separated from dissimilar samples. Unlike traditional classification, these methods are suitable for similarity search and zero-shot problems, where new and unseen classes appear during inference.

Siamese neural networks introduced the idea of learning similarity utilizing networks with shared weights [37]. Later approaches developed the concept using contrastive and triplet-based losses. A major development came with FaceNet [38], which demonstrated that Siamese neural networks trained using triplet loss could successfully learn semantically meaningful embeddings for large-scale similarity problems. They also showed that sampling is important for learning for these networks, introducing semi-hard triplet mining. Further work improved triplet-based learning through optimized batching strategies such as PK batching [39]. These methods

form the basis for the embedding learning approaches used in this thesis.

3.5 Similarity-Based Malware Analysis

Similarity-based malware analysis is a field of malware analysis that does not focus on detection and classification, rather on similarity between samples. The aim is to generalize to previously unseen malware samples and provide a deeper analysis of shared behavior. It does not necessarily need to involve deep learning, for instance call graph similarity [40] and fuzzy hash methods [41] have been used in previous research.

In recent work, embedding-based similarity methods have become relevant, often using metric learning [42], focusing on low-dimensional and computationally effective embeddings to reduce the storage capacity and computation needed for analysis. However, the problem of having no unified benchmarks and comparable similarity methods is challenging [43]. To fill that gap, EMBERsim was introduced, adding similarity tags to the data [17], as well as a leaf similarity method. Despite recent progress, significant challenges remain regarding standardized evaluation, comparable datasets, and generalization to unseen malware families.

3.6 Research Gap

Existing malware analysis research has largely focused on detection and classification using static and dynamic representations. While hybrid approaches have shown promising results for classification, relatively little work has explored similarity-based embedding spaces for retrieval, clustering, and zero-shot generalization.

Furthermore, direct comparisons between existing studies remain difficult due to differences in datasets, features, and evaluation. Many approaches ignore the zero-shot problem by only evaluating on seen families, limiting conclusions regarding generalization and real world applications.

Because for real world applications, malware analysis increasingly requires scalable methods capable of identifying semantic similarity between previously unseen malware samples. This creates a need for embedding-based malware representations that utilize static and dynamic representations, supporting similarity search and zero-shot generalization.

4
Theory

This chapter introduces the theoretical background relevant to the thesis, including static and dynamic malware analysis, metric learning, Siamese neural networks, transformer architectures, retrieval methods, and evaluation metrics. These concepts form the foundation for the experiments presented later in the thesis.

4.1 Static Malware Analysis

Static malware analysis extracts information from malware without executing them, in contrast to dynamic analysis that uses runtime behavior. Static analysis is a comparatively cheap method since the sample is never executed. The method is based on disassembly, decompilation, and reverse analysis of the binary to understand the execution characteristics. This method is widely used for malware detection and classification, both in academia and industry. Since machine learning cannot be directly applied to binary files in most cases, features are extracted from the binaries. Common features extracted from static analysis include PE header files, strings, control flow diagrams and API calls, among others, which enables machine learning models to be run, but the method struggles with complex malware using obfuscation and similar counter-detection measures [44].

4.1.1 PE Files

This thesis focuses only on Windows Portable Executable (PE) files, the standard architecture independent file format used to execute code on Windows. PE files, which contain all code and metadata to run a file, are binary files that are not human readable and are only meant for a machine to run. An in-depth knowledge of PE files is not necessary to understand this thesis, but some general knowledge is helpful. A PE file consists of several sections like library imports, API calls, program structure etc. [45]. Both malware and goodware use this format for their executables, and when a malware PE file is executed, the system is infected.

The PE file contains everything needed for an attack. As such, it is of great importance not to run suspicious files on a normal machine, and that is why static analysis is necessary. Information can be extracted from a PE file without running it, and static analysis often tries to detect or classify malware based on these files. It is often the case that API calls, imported libraries, etc. can indicate if a file is malicious or not, but threat actors of course know that their binaries will be analyzed,

and methods like packers and obfuscation is commonly used to trick these analyses [46].

4.1.2 EMBER Features

One widely used feature set for static analysis in academia is the EMBER feature set, introduced in 2018 [4]. It filled a void in machine learning applied to malware, namely a large dataset containing both malware and benign samples, and a standardized method to extract features from any PE file that can be used for machine learning tasks. EMBER features can be divided into eight groups of raw features, that are shown in Table 4.1.

Table 4.1: Overview of the EMBER feature categories and the information they encode. The extracted raw features are vectorized before being used as input to machine learning models [4].

Feature category	**Description**
General file information	File metadata such as file size, signatures, and debug-related indicators.
Header information	PE header metadata including system information, image versions, and timestamps.
Imported functions	Imported functions and linked libraries referenced by the executable.
Exported functions	Functions exported by the executable.
Section information	Section-level attributes such as names, sizes, entropy values, and permissions.
Byte histograms	Frequency distribution of byte values within the binary.
Byte entropy histograms	Approximate entropy distributions computed over byte sequences.
String information	Statistics and distributions of printable strings extracted from the binary.

As for all static analysis, the feature categories in Table 4.1 do not capture behavior, rather statistics as one can see. That is because behavioral information is not directly observable from a binary alone, and it shows the limitations of using static analysis in order to analyze behavior, which is ultimately the end goal for all cybersecurity analysis in industry; to understand what the malware will do if it is executed in a system. And in order to convert these raw features into numerical features needed for most ML-models, more information has to be removed. All string features, like function imports and exports, are put into hash buckets by modular hashing.

For individual functions for example, this means that the function name (k) is hashed ($h(k)$), and the modulo operator of the number of hash buckets (m) is applied ($\mathrm{mod} m$), which adds to the feature of index $i = h(k) \bmod m$. This method is applied to all non numerical raw features, in order to transform them into vectorized features that are used as input to ML-models. This is the trade off of using feature extraction models, information will be lost and only statistical features are stored, but by using EMBER features one can extract a 2381-dimensional vectorized feature vector from any binary, which enables traditional ML-models to be applied in static malware analysis. See Appendix A for raw EMBER features of a malware sample.

4.2 Dynamic Malware Analysis

Dynamic malware analysis consists, as stated above, of analyzing runtime behavior. This is done by running a so called “sandbox explosion” in tools such as Recorded Future’s Triage platform. To put it simply, this entails running potential malware samples on a contained and separate virtual machine, and then analyzing how that malware affects and interacts with the machine. While dynamic analysis is naturally a much more computationally expensive process than analyzing static modalities of a malware, it makes it more difficult for threat actors to obfuscate the behavior and purpose of a malware sample. Therefore, it is an additional tool that can be used when trying to understand malware.

4.2.1 Dynamic Reports

Due to the usually enormous volume of processes happening when running a “sandbox explosion”, the dynamic analysis is normally presented in a dynamic report. This report contains and outlines important characteristics and behaviors of a potential malware sample, which can then be analyzed in detail by experts to find patterns across malware. This in turn is one of the tools organizations and malware experts use to try to understand, compare, and categorize malware.

A dynamic report consists of several different sections describing different information, but only a subset are used in this thesis. The selection is based on previous research on dynamic malware analysis, which identifies several types of information as useful for malware classification and comparison [5, 10, 11, 31]. It should be noted, however, that sandbox environments differ between tools and organizations. This means that for the information available in the dynamic reports used in this thesis, the selected parts were chosen to approximate the types of information used in previous research. An outline of different parts can be seen in Table 4.2.

Table 4.2: Dynamic report sections used in this thesis.

Report section	Description
Network	Describes network-related behavior observed during execution, such as connections, DNS requests, HTTP traffic, protocols, ports, and transferred data.
Process	Describes process-related behavior, including created processes and interactions between processes.
Analysis	Contains general metadata and summary information about the sandbox execution and the sample.
Signature	Contains behavioral signatures triggered during execution, which summarize actions observed in the sample.
Dumped	Describes files or memory objects dumped during execution of the sample.
Extracted	Contains information about various infrastructure elements that are used by the malware.

4.3 Metric Learning

Metric learning is a machine learning approach which focuses on learning representations of inputs so that the similarity or dissimilarity between them can be measured meaningfully [47]. This is done by fulfilling the overarching training objective of reducing the distance between the similar inputs and increasing the distance between dissimilar inputs, according to a chosen distance or similarity metric.

This thesis focuses on how deep neural networks specifically can be used for metric learning task, also called deep metric learning. In this domain, the neural network architecture determines how input samples are embedded, while the loss function determines how the embedding space is structured during training; with the goal of achieving the aforementioned training objective.

4.3.1 Siamese Networks

A common neural network architecture used within metric learning are so called Siamese networks [48]. A Siamese network consists of two or more identical branches that process different inputs using the same neural network model. In other words, each branch shares the same weights, ensuring that all inputs are mapped onto the same embedding space which means embeddings can be compared to each other.

4.3.2 Triplet Loss

Triplet loss is commonly used in metric learning, which as the name suggests consists of processing triplets of samples through a shared embedding (Siamese) model and comparing the embeddings. Previous metric learning objectives encouraged samples from the same class to be projected onto one single point in the embedding space. Triplet loss, however, encourages samples from the same class to occupy a manifold of the embedding space [38]. This allows meaningful intra-class variation and information to be preserved, while the loss simultaneously encourages separation and discriminability to samples belonging to other classes.

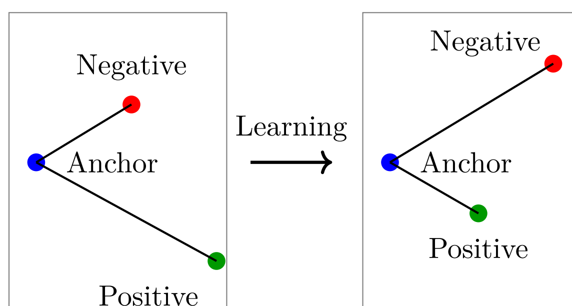


Figure 4.1: Illustration of the triplet-loss learning objective. During training, the embedding model learns to move positive samples closer to the anchor while simultaneously increasing the distance to negative samples in the embedding space.

Mathematically, triplet loss is based on triplets of samples. Each triplet consists of an anchor, a positive sample which belongs to the same class as the anchor, and a negative sample which belongs to a different class. The goal is for the distance between the negative and the anchor to be larger than the distance between the positive and the anchor plus a certain margin. This is illustrated in Figure 4.1.

Triplet-loss training therefore consists of two main components. First sampling a suitable set of triplets, and second calculating the loss of said triplets. The sampling is the difficult part of the process as ensuring fast convergence means finding suitable triplets that the model can learn from. One could mine hard negatives which are closer to the anchor than the positive, but FaceNet argued that those samples often lead to poor convergence [38]. The paper therefore introduced *semi-hard negatives*, samples that, as shown in Figure 4.2, are further from the anchor than the positive sample, but closer than the distance between the anchor-positive pair plus the margin.

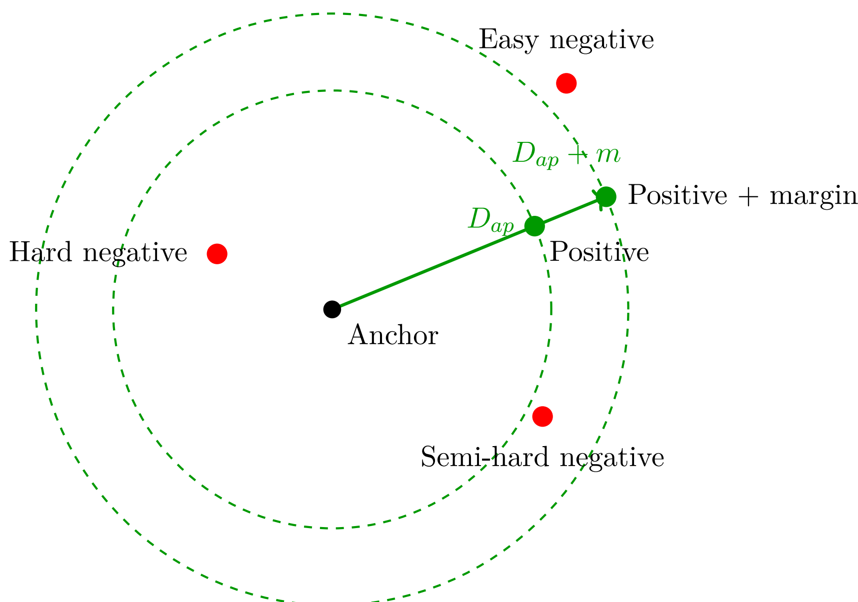


Figure 4.2: Illustration of triplet-loss sampling. Hard negatives lie closer to the anchor than the positive sample, semi-hard negatives lie outside the positive distance but within the margin, and easy negatives lie outside the margin boundary.

This thesis uses a slightly modified triplet-loss in comparison to how the triplet-loss is proposed in the FaceNet paper. Mathematically this can be described by the following using a batch $\mathcal{B}$ with inputs

$$\mathcal{B} = \{s_i\}_{i=1}^{B}, \tag{4.1}$$

where s_i denotes input sample i, and each sample has an associated class label y_i. The input sample may be a token representation from a sequence in the case of a transformer encoder, or a fixed-dimensional feature vector in the case of an MLP. In either case, the Siamese embedding model f_θ maps each input sample to an embedding vector e_i in the d_{emb}-dimensional embedding space $\mathbb{R}^{d_{\text{emb}}}$

$$e_i = f_\theta(s_i), \qquad e_i \in \mathbb{R}^{d_{\text{emb}}}. \tag{4.2}$$

Like in the FaceNet paper, each embedding is normalized before the loss is computed

$$z_i = \frac{e_i}{\|e_i\|_2}, \qquad \|z_i\|_2^2 = 1. \tag{4.3}$$

This means that the embeddings are mapped to a high-dimensional hyper-sphere, where the model encourages different clusters to map to different manifolds on the hyper-sphere. For all samples in the batch, pairwise Euclidean L_2 distance is calculated. Since the embeddings are normalized, the pairwise distance becomes

$$D_{ij} = \|z_i - z_j\|_2^2 = 2 - 2\cos(\theta_{ij}), \tag{4.4}$$

where θ_{ij} is the angle between the two normalized embeddings. This means that the distance measured for each sample pair is equivalent to the angular spread between the samples, and that the loss promotes the clusters to be angularly spread on the hyper-sphere.

After all samples in the batch have been mapped to embeddings and pairwise distances have been calculated, possible triplets are constructed using the original class labels y_i. This is done by letting each sample in the batch of size B be used once as an anchor. For each anchor sample $a \in \mathcal{B}$, the valid positives p are the other samples in the batch with the same class label y_a, while the valid negatives n are the samples with a different class label. These sets are defined as

$$\mathcal{P}_a = \{p \in \mathcal{B} \setminus \{a\} : y_p = y_a\}, \qquad \mathcal{N}_a = \{n \in \mathcal{B} \setminus \{a\} : y_n \neq y_a\}. \tag{4.5}$$

Here, $\mathcal{P}_a$ and $\mathcal{N}_a$ describe the sets of potential positive and negative indices, respectively, for anchor a in a batch of size B. For each anchor a, an anchor-positive pair (a, p) is formed for every positive sample $p \in \mathcal{P}_a$. Meaning an anchor a gives rise to $|\mathcal{P}_a|$ anchor-positive pairs. For each such pair, one negative sample $n^*(a, p)$ is selected from the anchor's negative set $\mathcal{N}_a$. The first choice is the closest negative $n^*(a, p)$ that is farther away from the anchor than the positive sample such that

$$n^*(a, p) = \arg\min_{n \in \mathcal{N}_a} D_{an} \quad \text{subject to} \quad D_{an} > D_{ap}. \tag{4.6}$$

Meaning that easy negatives can also be sampled. This follows the FaceNet idea, where if the negative lies inside the bounds

$$D_{ap} < D_{an^*} < D_{ap} + m, \tag{4.7}$$

that sample contributes to the loss. The difference to FaceNet is that when only easy negatives exist, meaning that

$$D_{an^*} \geq D_{ap} + m, \tag{4.8}$$

the triplet contributes zero loss, but the anchor-positive pair is still a part of $|\mathcal{P}_a|$. If no negative satisfies $D_{an} > D_{ap}$ for the specific anchor-positive pair (a, p), then

all negatives for that anchor are closer to the anchor than this particular positive sample. In this case, in order to still promote training, the implementation falls back to the easiest hard negative

$$n^*(a,p) = \arg\max_{n \in \mathcal{N}_a} D_{an}. \tag{4.9}$$

In practice, this situation is very unusual, since for sufficiently large batch sizes it is very unlikely that all negatives are closer to the anchor than the positive sample in the embedding space. Finally, the loss for the batch is calculated as

$$\mathcal{L}_{\text{batch}} = \frac{1}{|\mathcal{P}|} \sum_{a \in \mathcal{B}} \sum_{p \in \mathcal{P}_a} \Big[m + D_{ap} - D_{an^*(a,p)} \Big]_+ . \tag{4.10}$$

Where $|\mathcal{P}| = \sum_a |\mathcal{P}_a|$. Meaning that for each anchor a, the loss is summed over all anchor-positive pairs (a, p) using the negative $n^*(a, p)$ selected for that pair. The resulting losses are summed again and normalized by the total number of valid anchor-positive pairs, $|\mathcal{P}|$. This means that anchor-positive pairs with zero-loss, having only easy negative samples, are taken into account when scaling the loss.

The benefit of this loss is that it is computationally efficient and comparatively conservative, which is useful for malware similarity where family labels may be noisy, mislabeled, or behaviorally ambiguous. By prioritizing semi-hard negatives, using easy negatives as training converges, and relying on the easiest hard negative only as a fallback, the method reduces the influence of extreme or potentially unreliable triplets. However, this conservative behavior risks not using meaningful information in hard negatives, although it is highly unlikely that a sample would not be part of a semi-hard triplet over the entire batch.

4.3.3 *PK* Batch Sampling

PK batch sampling is the process of sampling K samples each from P classes for one batch $B = PK$. In the case of malware, this means sampling P malware families and K samples per family. This type of sampling is very important in triplet-loss as the convergence of the model depends on it seeing enough valid triplets in each batch. Using triplet loss on a PK batch, the number of positive and negative samples for each anchor are

$$|\mathcal{P}_a| = K - 1, \qquad |\mathcal{N}_a| = (P-1)K. \tag{4.11}$$

Where the valid negatives are all samples K samples belonging to each of the remaining $P - 1$ families. For the modified triplet loss in Equation 4.10, each sample in a batch $\mathcal{B}$ is used once as an anchor, meaning that the total number of anchors to walk through is $B = PK$. Therefore the amount of anchor-positive pairs in a batch is:

$$|\mathcal{P}| = \sum_{a \in \mathcal{B}} |\mathcal{P}_a| = PK(K-1). \tag{4.12}$$

This means that the loss function in Equation 4.10 is normalized by a fixed number for each batch. A benefit of this approach is that one can ensure a certain amount of valid triplets exists in each batch. This decreases the need of computationally heavy large batches, which are needed without PK-sampling to ensure that there are triplets to learn from.

4.4 Transformers Applied to Metric Learning

This section introduces the transformer concepts necessary for understanding the transformer embedding models used in this thesis. In particular, the focus is on the self-attention mechanism and how transformers can be used to learn representations for metric learning and malware similarity tasks.

4.4.1 The Basis of Transformers

Transformers are a neural network architecture introduced in the influential "Attention Is All You Need" paper [1]. They were originally developed for natural language processing tasks such as machine translation, but are now widely used across many machine learning domains. Unlike recurrent neural networks, transformers process entire sequences in parallel using an attention mechanism that allows each token to incorporate information from other tokens in the sequence.

The key component of the architecture is self-attention, where each token attends to other tokens in the same sequence in order to form a contextualized representation. Intuitively, self-attention allows each token to determine which other parts of the sequence are most relevant when constructing its representation. In self-attention, the same input sequence is used to construct the query, key, and value representations. Mathematically, the input sequence is represented as a matrix

$$X \in \mathbb{R}^{n \times d_{\text{model}}},$$

where n is the sequence length and d_{model} is the embedding dimension. Learned linear projection matrices are then used to construct query, key, and value representations

$$Q = XW^Q, \qquad K = XW^K, \qquad V = XW^V, \tag{4.13}$$

where

$$W^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, \qquad W^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \qquad W^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$$

are learned projection matrices for the query, key and value sequences. Here, d_k denotes the dimensionality used for the query and key projections, while d_v denotes the dimensionality of the value projections. The attention output for a single attention head is then computed as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V. \tag{4.14}$$

The attention matrix $S = \frac{Q \cdot K^\top}{\sqrt{d_k}}$ contains all pairwise query-key dot products, where element (i, j) measures how strongly token position i attends to token position j, scaled by the key dimension d_k to improve gradient stability during training.

The softmax operation normalizes each row of the attention score matrix into attention weights that sum to one. The resulting matrix elements are then used to form weighted sums of the value vectors in V. Multi-head in this case means that several attention heads perform this operation several times in parallel using different learned projections, after which the results are concatenated and linearly projected back to the model embedding dimension. This projected attention output is in the original paper combined with the original input through a residual connection and is followed by layer normalization. The result is then passed through a feed-forward network, again followed by a residual connection and a normalization step. See Figure 4.3 to see the entire architecture.

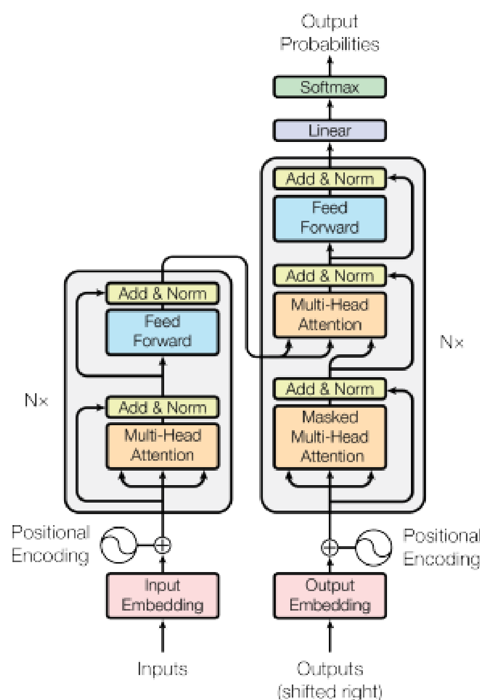


Figure 4.3: A picture of the original transformer architecture. Image taken from the original "Attention Is All You Need" paper [1].

Note that modern transformers repeat this attention-to-feed-forward block over several layers, so that the final layer produces a highly contextualized embedding vector for each token. This embedding is then used differently depending on the task. Other sources that give a visual explanation of transformers and the attention mechanism can be found in [49, 50]. Note, in the standard encoder structures, attention allows each input token to attend to all other tokens, including itself, in the input sequence.

4.4.2 BERT

The Bidirectional Encoder Representations from Transformers (BERT) is a transformer architecture developed by Google [51]. It is based on the Transformer encoder architecture, meaning that it processes an input sequence using attention and produces contextualized vector embedding representations for each token in the sequence. This embedding space can then be used for downstream tasks, primarily for retrieval and classification within the natural language processing domain. A visual explanation of how it works can be seen in Figure 4.4.

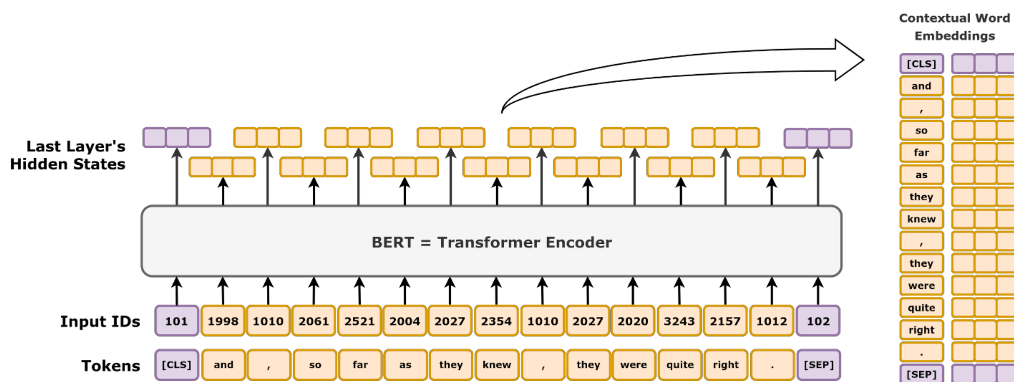


Figure 4.4: This figure shows conceptually how an input sequence of tokens is encoded through BERT. It takes the sequence, adds certain special tokens such as [CLS] and [SEP] to signify start and stop of a sequence, and finally creates embedding vectors for all tokens in a sequence. The [CLS] token is typically used to represent the full sequence. Image by Daniel Voigt Godoy, licensed under CC BY 4.0 [2].

The term bidirectional refers to the fact that, in contrast to left-to-right language models such as openAI's original GPT models, BERT allows each token representation to be attended by information from both previous and future tokens in the sequence in order to improve contextual understanding of unlabeled text. In other words, the embeddings of a token can be influenced by context on both its left and right side in a sequence. To train BERT, the paper specifically introduced using the masked language modeling (MLM) training objective. It is a self-supervised method which works by, as the name suggests, masking with a certain probability a subset of selected input token where the model is then forced to try to predict which word was there originally based on the bidirectional context.

To describe MLM mathematically, let $x = (x_1, x_2, \ldots, x_n)$ denote the original token sequence, let M denote the set of selected masked positions, and let $\tilde{x}$ denote the masked input sequence. For each masked position $i \in M$, BERT produces a contextualized hidden representation h_i by passing $\tilde{x}$ through the encoder. This hidden representation is then passed through a feed-forward prediction head, producing a logit vector $z_i \in \mathbb{R}^{|\mathcal{V}|}$ over the vocabulary. The probability assigned to a vocabulary token $v \in \mathcal{V}$ is obtained by applying the softmax function:

$$P_\theta(x_i = v \mid \tilde{x}) = \frac{\exp(z_{i,v})}{\sum_{u \in \mathcal{V}} \exp(z_{i,u})}, \tag{4.15}$$

where $\mathcal{V}$ is the vocabulary, z_i is the logit vector for the masked position i, and $z_{i,v}$ denotes the logit corresponding to vocabulary token v. These probabilities are then used to generate a loss by using the cross-entropy loss function:

$$\mathcal{L}_{\text{MLM}} = -\frac{1}{|M|} \sum_{i \in M} \log P_\theta(x_i \mid \tilde{x}). \tag{4.16}$$

That is, the training objective is for the model to produce a probability distribution where the actual original token at the masked position has a high probability. A picture illustrating MLM can be seen in figure 4.5.

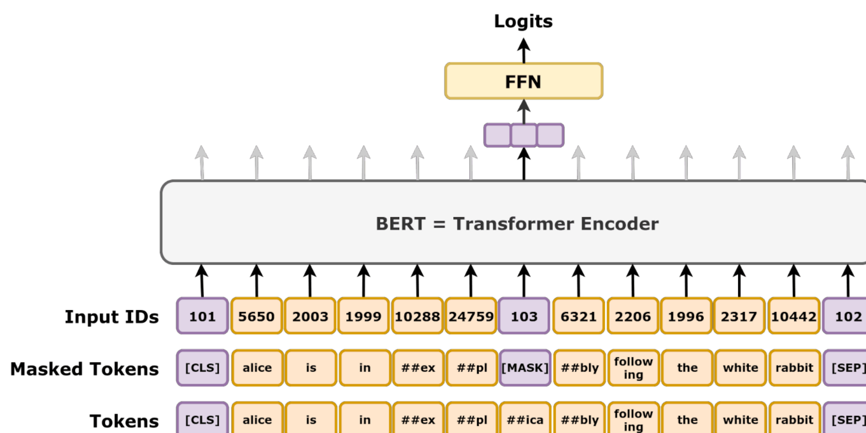


Figure 4.5: This figure shows conceptually how the BERT model is trained using MLM. A certain amount of tokens are masked, as shown by the [Mask] symbol, and the objective is for the model to predict which word was there originally based on the bidirectional context. Notice how the masked token is fed through BERT to produce a hidden representation, then through a feed-forward network to produce logits. Image by Daniel Voigt Godoy, licensed under CC BY 4.0 [3].

What makes a BERT model so powerful is its versatility. As the original paper shows, pre-training the BERT model using for example MLM, allows for several configurations and downstream tasks by using the learned contextualized embedding space. One can then fine-tune it for a new task by swapping out inputs, outputs and training objectives [52, 53, 54, 55].

The BERT configuration used in this thesis is ModernBERT, a modernization of the original BERT methodology and architecture to improve downstream performance and efficiency, especially over longer sequence lengths [56]. It is trained on 2 trillion tokens and implements several empirically-derived improvements in the encoder architecture based on previous research on BERT-like models. The details of these changes are not important for this thesis, but rather the outcome of them. The improvements in efficiency allow ModernBERT to drastically expand its context window, so it can process more tokens at the same time. It can process sequences of

8192 tokens almost two times faster than previous models. For more information on the details, see this blog post from the original authors of the ModernBERT paper explaining their methodology and results in a more visual manner [57].

4.4.3 Sentence Transformer

Sentence-BERT (SBERT) extends BERT through metric learning, by using a Siamese network architecture to produce sentence embeddings that can be compared directly using similarity measures such as cosine similarity [58]. In the original BERT cross-encoder setup, two input sequences are concatenated and processed jointly through the transformer in order to predict their similarity. While this enables strong pair-wise reasoning, it requires recomputing the transformer output for every sequence pair, making large-scale similarity search computationally expensive. SBERT instead independently encodes each sequence using a shared Siamese BERT encoder, after which similarity can be computed directly between the resulting embeddings.

To do this, for an input sequence s_i, BERT produces one contextualized hidden representation for each token in the sequence, as illustrated in Figure 4.4. If the sequence contains n_i tokens, the encoder output can be written as

$$H_i = (h_{i1}, h_{i2}, \ldots, h_{in_i}), \qquad h_{ij} \in \mathbb{R}^d.$$

Here, i denotes the input sample and j denotes the token position within the sequence of that sample. The token-level representations are then fused into a fixed-size sentence-level embedding $S_i \in \mathbb{R}^d$, for example using mean pooling:

$$S_i = \frac{1}{n_i} \sum_{j=1}^{n_i} h_{ij}. \tag{4.17}$$

SBERT can be fine-tuned using different objectives. One of them is using a triplet loss objective. What this means is that the sentence embedding for each triplet sample is passed through a Siamese BERT architecture, which is then used to generate a triplet loss using Equation 4.10. The loss is then used to fine-tune the BERT model to place semantically similar inputs close together in the embedding space.

The main advantage of SBERT paired with a triplet-loss training objective, is that the resulting embeddings can be computed independently and compared efficiently. It also has a faster training time compared to other randomly initialized methods used for sentence similarity clustering and retrieval, due to using a pre-trained BERT model and simply fine-tuning it.

Due to the versatility of this approach, the same methodology can also be applied using ModernBERT. This combines a large transformer encoder with an extended context window, which comes with the contextual representation benefits of the attention mechanism, and a metric learning objective. Together, these properties make it possible to process large and information-dense inputs while placing semantically related representations close in the embedding space. In this thesis, this is par-

ticularly relevant for dynamic malware reports, which are often long, semantically complex, and rich in information.

4.4.3.1 Report Pooling

To apply SBERT to the malware domain, one approach is to use it to embed dynamic malware reports. However, these reports can contain thousands, or even tens of thousands, of tokens. Moreover, as stated previously, different parts of a report may describe different aspects of the malware behavior. As a result, even when using an encoder-transformer model like ModernBERT with a long context window of 8192 token, a single sample's report may need to be divided into several smaller chunks of token sequences. Therefore, each chunk risks only containing a small fraction of the information needed to properly describe a sample's behavior.

This problem is exacerbated by the computational cost of long-context transformer models. In self attention, the attention computation scales quadratically with the sequence length [1]. So in reality, in order to maintain reasonable training time and to not run into out-of-memory errors, it may be necessary to use shorter chunks than the maximum context window supported by the model.

For metric learning objectives such as triplet loss, this creates a mismatch between chunk-level embeddings and the desired report-level representation of malware behavior. One approach to handle this is to aggregate chunk embeddings into pooled report-level representations during similarity learning. The exact implementation used in this thesis is described in the Methods chapter.

4.5 Retrieval and Reranking

A common problem with retrieval tasks in machine learning is that the more advanced methods are often computationally expensive. Therefore, a widely used approach is retrieval followed by reranking. The main idea is that a computationally inexpensive retrieval method, such as a nearest neighbor search in an embedding space, is first used to identify candidates similar to the query. Then a more refined method can be used on that subset to obtain a more accurate similarity estimation. This way, one does not need to scan the entire set using the computationally expensive method, while still achieving a more accurate similarity estimation than using only the embedding model.

4.5.1 k-Nearest Neighbors

k-Nearest Neighbors, kNN, is a widely used method for semantic retrieval. It is a method to extract semantically similar embeddings by finding the nearest embeddings to a query vector. Mathematically, the expression is simple,

$$\mathcal{N}_k(q) = \underset{x_i \in D}{\arg\operatorname{topK}} \ \operatorname{sim}(q, x_i), \tag{4.18}$$

where $\mathcal{N}_k(q)$ denotes the set of k-Nearest Neighbors for a query q, D is the embedding database, and $\mathrm{sim}(q, x)$ is the similarity function used, typically cosine similarity, see Equation 4.21 [59].

4.5.2 Best Matching 25

Lexical reranking is a method used to improve retrieval results by emphasizing exact keyword overlaps between the query and retrieved documents, a method often too computationally expensive to run directly on huge datasets. BM25 is a widely used lexical reranking method, with the core function to match documents with common terms. Embedding retrieval methods capture semantic similarity, while BM25 complements embedding-based retrieval by emphasizing exact lexical matches between query and document terms. This can improve retrieval precision in cases where specific terminology or keywords are important.

Mathematically, for a corpus of documents the BM25 score between a query q and a document d in the corpus is defined as

$$\mathrm{BM25}(q, d) = \sum_{t \in q} \mathrm{IDF}(t) \cdot \frac{f(t, d)(k_1 + 1)}{f(t, d) + k_1 \left(1 - b + b \frac{|d|}{\mathrm{avgdl}}\right)}, \tag{4.19}$$

where $f(t, d)$ is the term frequency for a term t from q in d, $\mathrm{IDF}(t)$ is the inverse document frequency of the term in the corpus, $|d|$ the document length, and avgdl the average document length in the corpus, and k_1 and b are tuning parameters, with typical parameter choices $k_1 = 1.2$–2.0 and $b = 0.75$.

Intuitively, increased frequency of a term from the query in the document increases the score, weighted with the rarity of the term in the corpus. It is also normalized for document length, to avoid bias towards longer documents.

4.6 Evaluation Metrics

To evaluate embedding and clustering results, one needs to use suitable evaluation metrics that can capture the quality of the embeddings. Since the embedding space often is high-dimensional, getting an overview of the embedding space is often impossible, without using scalar metrics or reducing dimensionality. Commonly, straight forward metrics like neighborhood purity, recall, and hit rate are used, but there are also more refined methods with a deeper mathematical background.

4.6.1 Davies-Bouldin Index

A widely used evaluation metric for class based clustering is the Davies-Bouldin Index. The measure indicates similarity within clusters, and does not depend on the numbers of clusters or the method used to create them. The index is lowered (improved) by increased inter-cluster distances and decreased intra-cluster distances

[60].

Given a finite number of points in $\mathbb{R}^n$, let A_i be the centroid and T_i the number of points for cluster i, then $S_i = \frac{1}{T_i}\sum_{j=1}^{T_i} \|X_j - A_i\|$ is the average Euclidean distance compared to the center of the cluster i and $M_{ij} = \left(\sum_{k=1}^{n} \|A_{ki} - A_{kj}\|^2\right)^{1/2}$ is the Euclidean distance between centroids of two clusters i and j. Then $R_{ij} = \frac{S_i+S_j}{M_{ij}}$ is defined as the cluster similarity between cluster i and j, and $R_i \equiv \max_{j\neq i} R_{i,j}$ is the cluster similarity of cluster i to the most similar other cluster. Then the Davies-Bouldin Index $\overline{R}$ for N clusters using Euclidean distance is defined as

$$\overline{R} = \frac{1}{N}\sum_{i=1}^{N} R_i. \tag{4.20}$$

4.6.2 t-SNE

As previously said, when working with high-dimensional embeddings, one cannot visualize them in an understandable way without reducing the dimensionality. To keep information like clusters, separation, etc. when doing this is not trivial. There are many different methods developed to do this like PCA and UMAP, but the one used in this thesis is t-SNE, t-distributed stochastic neighbor embedding, for its ability to visualize clusters and complex relationships in data [61]. It has its strengths when it comes to local neighborhoods and clusters, but global distances loose meaning, something that needs to be taken into account.

The core idea behind t-SNE starts with the conditional probability that given a set of points $x_1, ..., x_N$, what is the probability $p_{j|i}$ that x_j is chosen as neighbor of x_i given a Gaussian distribution centered around x_i, with a standard deviation σ_i,

$$p_{j|i} = \frac{\exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma_i^2}\right)}{\sum_{k\neq i}\exp\left(-\frac{\|x_i - x_k\|^2}{2\sigma_i^2}\right)},$$

with $p_{i|i} = 0$. The denominator is the sum of the probability for all points with regard to the Gaussian distribution for x_i, to remove the impact of different cluster densities. Then define the joint probability $p_{i,j} = \frac{p_{i|j}+p_{j|i}}{2N}$, which measures the similarity between the two points. The goal for t-SNE is to learn a mapping to $y_1, ..., y_N$ with $y \in \mathbb{R}^d$, with d typically 2 or 3, that reflects p_{ij} as well as possible. To do this, it measures similarity between two points y_i and y_j in a similar way, defining $q_{i|j}$ using the Student's t-distribution instead, for its heavy tails that can map moderate distances in the original embedding space to more extreme distances in $\mathbb{R}^d$ to reduce crowding. Then q_{ij} is defined as

$$q_{ij} = \frac{\left(1 + \|y_i - y_j\|^2\right)^{-1}}{\sum_{k\neq l}\left(1 + \|y_k - y_l\|^2\right)^{-1}},$$

with $q_{ii} = 0$. To make these distributions as similar as possible, Kullback-Leibler divergence, a measure of similarity between two probability distributions, is used [62]

$$KL(P\|Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}.$$

The embedding coordinates y_i are optimized by minimizing the Kullback-Leibler divergence using gradient descent, resulting in a low-dimensional representation reflecting the local similarities in the original high-dimensional data.

As the attentive reader may have noticed, σ_i has not yet been defined explicitly. The standard deviation of the Gaussian distribution is determined indirectly through the perplexity hyperparameter, which controls the effective neighborhood size around each point. Lower perplexity values, corresponding to smaller σ_i, emphasize local structure and smaller clusters, while higher values preserve more global relationships between points.

4.6.3 Cosine Similarity

When evaluating embeddings, cosine similarity is often preferred over distance metrics like Euclidean distance, since it measures angular similarity independently of vector magnitude. Mathematically, cosine similarity between two vectors x and y is defined as

$$\text{sim}(x, y) = \frac{x \cdot y}{\|x\|\|y\|}, \tag{4.21}$$

which corresponds to the cosine of the angle between the vectors. If embeddings are normalized, $\|x\| = 1$, then this corresponds to the Euclidean distance between the vectors like $\|x - y\|^2 = 2(1 - \text{sim}(x, y))$. This means that minimizing the Euclidian distance is equivalent to minimizing the angle between the vectors.

Cosine similarity measures the angular similarity between vectors and is therefore independent of vector magnitude. This makes it well suited for embedding spaces, where semantic similarity is commonly represented through the relative orientation of vectors.

5
Methods

This chapter describes the methodology used in this thesis. An overarching description is shown in Figure 5.1, which shows a visualization of the various steps taken throughout the thesis.

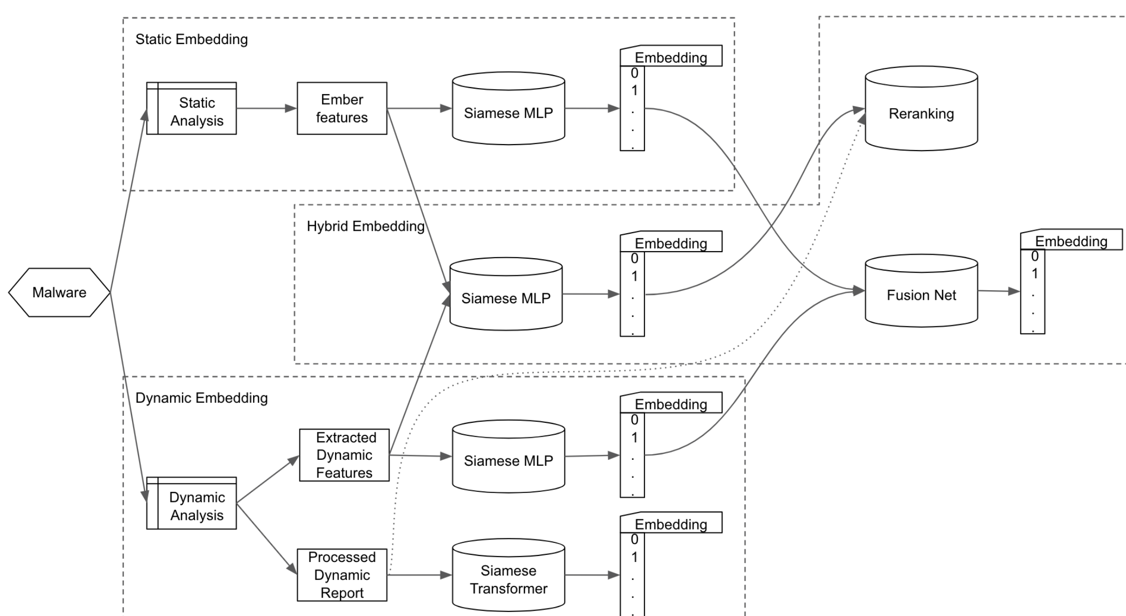


Figure 5.1: Overview of the malware embedding pipeline. Everything starts from the malware samples, that are analyzed both statically and dynamically. The malware is then processed, and fed to machine learning models used to create the embeddings. Both static and dynamic embeddings were created and analyzed. Hybrid embeddings were created both by feeding a model static and dynamic features, as well as fusing embeddings from static and dynamic models, and then analyzing them. In the case of dynamic models, both Siamese MLP and transformers are used to create embeddings. Note that the Siamese transformer in this case refers to BERT transformer encoders. A BM25 reranking method on the hybrid embeddings was also implemented and analyzed.

5.1 Data

The dataset used for the thesis consists of approximately two million malware samples from Recorded Future's internal malware analysis platform. Each sample is

associated with one family label and a unique Windows PE file. In total, 330 different malware families occur in the dataset.

The dataset is heavily imbalanced, which reflects the real world distribution. For instance the “Berbew” family has 591,647 samples in the dataset, while there is only a single “thunderkittystealer” sample. Such imbalance presents a challenge for machine learning methods, since large families risk dominating the learned embedding space. Another issue is that since the used supervised embedding models require family labels, no unlabeled malicious files are included. This is an issue for a model’s ability to generalize to real world data, but one that is difficult to overcome.

To include benign files in the experiments, benign PEs were obtained from the PE Malware Machine Learning Dataset created by Michael Lester [63]. It is a known issue in the malware analysis field that benign files are difficult to obtain at scale, since they fall under copyright regulations. Using this open source dataset solves that problem, but limits the experiment, since this is not a fully representative distribution of benign software. However, it is not feasible to obtain and train on a representative distribution of all benign files.

Although the malware dataset is not publicly released due to legal and operational constraints, the feature extraction methods used throughout the thesis are reproducible and based on publicly available methods. Static representations are extracted using EMBER features [4], which can be generated from any Windows PE file. Similarly, the dynamic representations are based on sandbox execution reports, and can be reproduced using other sandbox environments. Consequently, while the exact dataset cannot be publicly released, the preprocessing pipelines, feature extraction methods, and evaluation procedures are reproducible and can be applied to other malware datasets.

Lastly, it should be noted that family labels are not a perfect representation of malware. They are noisy, often ambiguous, and an imperfect description. Family labels are generated by rule based antivirus systems, and when families evolve over time through obfuscation and modification, family boundaries are unclear. As a result, they cannot perfectly measure similarity, but with limited alternatives, their approximation of similarity is used in the thesis.

5.1.1 Static Data

The static data used throughout the thesis consisted of raw Windows PE binary files, and no runtime behavior was used. Instead, all information was extracted directly from the binary files themselves.

5.1.1.1 Manual Feature Extraction

Static malware representations were extracted using the EMBER v2 feature extraction framework introduced in [4]. As described in Section 4.1.2, EMBER extracts statistical and structural information directly from PE binaries and converts them

into fixed-length numerical feature vectors suitable for machine learning applications. The resulting vectorized representations are 2381-dimensional.

5.1.1.2 Preprocessing

The preprocessing pipeline applied in this thesis extended beyond the standard EMBER feature representation. First feature selection was performed. EMBER consists of mostly statistical features and to avoid redundant finger-print-like information to be encoded, some feature categories were excluded. Feature selection for EMBER-based similarity learning remains relatively underexplored in the literature, particularly for metric learning applications. Metric learning objectives are particularly sensitive to artifact-driven shortcuts, since the embedding space is optimized directly through pairwise similarity relationships.

The biggest category that was excluded was imported functions, consisting of 1280 features in an EMBER feature vector. They were excluded because, these are hashed imported function names, forming a high-dimensional and sparse representation sensitive to obfuscation, since the imports are routinely renamed, or obfuscated in other ways. As a result, they may not form a reliable basis for similarity learning, instead encoding dataset specific artifacts. The same reasoning applies to the exported functions and section names categories, that were also excluded. Another category that was excluded was the categorical PE header category. These features reflect compiler, toolchains, etc. rather than malware behavior.

The remaining categories: byte distributions, string statistics, general file metadata, header numerics, section counts, and data directories, capture structural and statistical properties across malware variants, and were therefore chosen for the 672-dimensional feature vector. Those were then scaled differently according to category, to better separate differences in the input space, according to Table 5.1.

Histogram-based features were transformed using square root scaling followed by feature group-wise ℓ_2 normalization in order to reduce dominance from highly frequent byte values and emphasize distributions instead of magnitudes. Count- and size-heavy blocks use $\log(1 + x)$ scaling followed by z-score normalization (fit on training data) to reduce skew and make dimensions comparable. Continuous summary features like average string length and entropy only use z-score, while binary flags are left unscaled.

5.1.2 Dynamic Data

The dynamic data used throughout the thesis consisted of dynamic reports of runtime behavior. No features were extracted from the raw binaries.

Table 5.1: Selected EMBER feature groups and preprocessing pipeline used in this work. Excluded features include categorical header fields, detailed section attributes, import/export features, and other high-cardinality sparse metadata.

Feature group	**Preprocessing**	**Dim.**
Byte histogram	$\sqrt{x}$ + block ℓ_2 normalization	256
Byte-entropy histogram	$\sqrt{x}$ + block ℓ_2 normalization	256
String counts	$\log(1+x)$ + z-score	6
String summaries	z-score	2
Printable string histogram	$\sqrt{x}$ + block ℓ_2 normalization	96
General counts/sizes	$\log(1+x)$ + z-score	5
General binary flags	Raw binary values	5
Header versions	z-score	8
Header sizes	$\log(1+x)$ + z-score	3
Section summaries	$\log(1+x)$ + z-score	5
Data directories	$\log(1+x)$ + z-score	30
Total retained dimension		**672**

5.1.2.1 Manual Feature Extraction

Similarly to static EMBER features, the goal of this step is to produce a standardized feature set derived from the dynamic reports of each malware sample. These features are subsequently represented as numerical input vectors to be used in the machine learning models.

Part of the feature design is inspired by prior work on sandbox-based dynamic malware analysis [10, 11, 5]. The feature set is organized into four overarching categories: Network, Process, Memory Dumps, and Signatures, all of which are derived from dynamic analysis reports. Based on these categories, universal dynamic feature sets were constructed that could be applied consistently across all samples. This ensures that every sample is represented in the same dimension, resulting in a feature vector of 622 dynamic features per sample. In the subsequent sections, these categories and their associated features are described in more detail.

Network Features

As described in Table 4.2, network features describe network activity observed during a sandbox “explosion” of a malware sample. The important information includes everything from the size of the network traffic, to which communication methods a malware utilizes. Traditionally, this type of information is encoded in packet capture (PCAP) files, which is not part of this thesis dataset. Luckily, the dynamic reports contain much of the same information.

The reason why these types of features were used is because it is much harder to hide and obfuscate actual network behavior and activity compared to static characteristics [10]. In theory, this makes these features good candidates for capturing some behavioral information.

As a context to these features, it is useful to briefly define the relevant parts of the

network section in the dynamic reports. The most important components are flows and requests. Flows describe communication between IP addresses, including transmitted data volumes, transport protocol, and ports. Requests provide more detailed information about domain name systems (DNS), TLD, and hypertext transfer protocol (HTTP) activity, including request types and responses. A complete overview of the extracted network features is given in Table B.1 in Appendix B.

Signature Features

Signature features describe a set of behavioral signatures that are seen during a sandbox execution. The main signatures used are Windows system API calls made by the malware. These are calls where the malware interacts with the operating system, for example to access files, create processes, modify the registry, allocate memory, or perform network operations. However, the dynamic reports used in this work do not contain the raw API-call sequences. Instead, they contain more higher-level behavioral categories that describe the actions observed during execution, which are then used as a feature category.

Moreover, the reports also contain tactics, techniques, and procedures (TTPs) identified during execution. These TTPs are described using the MITRE ATT&CK framework, a knowledge base maintained by MITRE in which malicious behavior is categorized into TTP codes [64]. A single malware sample or family may therefore be associated with several TTPs, depending on the behaviors observed during sandbox execution. A more thorough description can be found in table B.2 in Appendix B.

Process Features

Process features describe how processes are created and the interaction between different processes during execution. This includes created processes, executable image paths, command-line arguments, files used and parent-child process relationships. Table B.3 in Appendix B describes these features in detail.

Memory dump Features

Finally, memory dump features are the conceptually easiest features as they simply describe the different sizes and amounts of files or memory objects dumped during the sandbox execution. Table B.4 in Appendix B describes these in detail.

5.1.2.2 Preprocessing

The scaling was done by removing the mean of the training data distribution and scaling to unit variance. This led to a Gaussian distribution with zero mean and unit variance for each feature before feeding them to the model. This scaler was then used when inferring embeddings.

5.1.3 Hybrid Data

Hybrid malware representations were constructed by combining static and dynamic feature vectors. Two different hybrid approaches were investigated in this thesis. The first approach used feature-level fusion, where the vectorized static EMBER features and the manually extracted dynamic features were concatenated into a single hybrid feature vector prior to training. The same preprocessing and scaling procedures described previously for the individual modalities were applied before concatenation.

The second approach used embedding-level fusion. In this setup, separate static and dynamic embedding models were first trained independently. The resulting embeddings from the two models were then concatenated to form a combined hybrid representation, which could then be used as input to a fusion network. In theory, feature-level fusion allows a model to jointly learn relationships between static and dynamic features directly from the input space, while embedding-level fusion instead combines the embeddings learned independently from each modality.

5.1.4 Preprocessing of Dynamic Reports

The dynamic reports needed to be preprocessed in order to reduce the computational resources needed to use them for transformers and BM25 reranking, but more importantly to get rid of data leakage, since the reports include analysis results.

5.1.4.1 Transformer Encoders

The transformer encoders use textual representations of the dynamic reports as input. The preprocessing step therefore prepares the dynamic report text, instead of constructing fixed size feature vectors. This preprocessing consists of two main steps: data cleaning and normalization. This is heavily inspired by the work done in “Nebula: Self-Attention for Dynamic Malware Analysis” [31], where the authors built a transformer encoder classifier from scratch. However, since this thesis uses pre-trained BERT-based architectures, the preprocessing steps differ from Nebula, which built a tokenizer and vocabulary from scratch. This is because BERT models, such as ModernBERT, have already been trained on a massive training corpus of natural language text and have a large vocabulary, and changing the tokenizer or vocabulary would reduce the benefit of using these pre-trained models.

More specifically, the data cleaning step consists of filtering out the unnecessary parts of the dynamic reports and organizing each remaining part in a clearly structured way. The fields used for the transformers are the ones listed in Table 4.2, which represent the most semantically important and discriminative parts of the report. Example of what these parts can look like can be seen in the listings of report excerpts C.1, C.2, C.3, C.4, C.5 and C.6 found in Appendix C.

As shown by the report excerpts in Appendix C, these sections are highly information-dense, irregular and contain a lot of obfuscated information of little semantic mean-

ing. Moreover, many of these fields change for every new run done in a sandbox execution due to obfuscation attempts by threat actors. This makes normalization an important preprocessing step, as the purpose of it is to remove or generalize information that is unlikely to be semantically useful, while preserving information that describes the underlying behavior of the sample. For example, an individual IP address may not be meaningful in itself, since the infrastructure used for a malware is frequently changed by threat actors. However, whether the address is public, private, loopback, or belongs to another category may still provide useful behavioral information.

This thesis normalizes several types of report values and fields. Hash values, including MD5, SHA1, and SHA256 hashes, are replaced with placeholders such as <md5>, <sha1>, and <sha256>. IP addresses are mapped to placeholders describing their address type, such as for example private, public, loopback, or IPv6 addresses. Domain names are replaced with a general <domain> placeholder. Common Windows path patterns are normalized using placeholders such as <drive> and <user>. Report-specific artifacts, such as memory addresses, dump paths, and automatically generated object names, are also normalized using placeholders throughout the report, in order to reduce run-specific noise. Finally, explicit malware family names are replaced with the generic placeholder <family>, which is done in order to avoid data leakage. An example of what a normalized report looks like after these steps, can be seen in Listing 5.1.

Listing 5.1: Excerpt from one cleaned and normalized dynamic report showing representative entries from the network, process, analysis, signature, dumped, and extracted sections.

```
Network:
{’id’:2,’src’:’<private>:61844’,’dst’:’<public>:53’,’proto’:’tcp’,
 ’pid’:1912.0,’procid’:34.0,’first_seen’:1601.0,’last_seen’:1985.0,
 ’rx_bytes’:398,’rx_packets’:5,’tx_bytes’:327,’tx_packets’:6,
 ’protocols’:array([’dns’],dtype=object),
 ’domain’:’<domain>.com’}

...

{’flow’:2,’index’:1,
 ’dns_request’:{’domains’:[’<domain>.com’],
 ’questions’:[{’name’:’<domain>.com’,’type’:’IN A’}]}}

...

(’<public>’,{’cc’:’IN’,’asn’:’AS8075’})

Process:
{’procid’:83,’procid_parent’:56.0,’pid’:736,’ppid’:3432,
 ’cmd’:’<drive>\\users\\<user>\\<temp>\\<sha256>.exe’,
 ’started’:125,’terminated’:1141.0}

...

{’procid’:86,’procid_parent’:84.0,’pid’:1864,’ppid’:800,
 ’cmd’:’<drive>\\users\\<user>\\<appdata>\\omsecor.exe’,
 ’started’:391,’terminated’:1047.0}

Analysis:
{’tags’:[’discovery’,’trojan’]}
{’ttp’:array([’T1614.001’],dtype=object)}

Signatures:
{’name’:’Suspicious use of WriteProcessMemory’,
 ’indicators’:[{’description’:’PID 736 wrote to memory of 800’,
 ’pid’:736.0,’procid’:83.0,’procid_target’:84.0}]}

Dumped:
{’at’:157,’pid’:736,’procid’:83.0,’name’:’<memory_dump>’,
 ’kind’:’region’,’origin’:’exception’,
 ’addr’:’<memory_address>’,’length’:147456.0}
```

```
...

{'at':469,'pid':1864,'procid':86.0,
 'path':'<drive>\\users\\<user>\\<appdata>\\omsecor.exe',
 'name':'<file_dump>','kind':'martian',
 'sha256':'<sha256>','size':137285.0}

Extracted:
{'config':{'c2':['http://<domain>.com/',
                 'http://<domain>.net/']}}
```

Moreover, each dynamic report needed to be tokenized in order to be inputted into the transformer encoder. However, since most reports had more tokens than the model's context window, the reports had to be split into shorter chunks of a certain sequence length, before being passed to the transformer. These chunks were created by tokenizing a report through the model's built-in tokenizer, splitting the resulting token sequence into sequence length chunks, and then decoding each segment back to text. The main evaluated sequence length for Siamese training was $N = 1026$ and 512 for MLM for the ModernBERT.

5.1.4.2 BM25

For the BM25 reranking step, a different preprocessing strategy was used. While the transformer encoder models require preprocessing that supports generalization across reports, BM25 relies on lexical matches and term frequency within the corpus being compared. Therefore, the preprocessing is less focused on abstraction and normalization, and instead aims to preserve the concrete textual details found in the reports in Appendix C. The same parts of the report that the transformers used, are also used for BM25 reranking. The only normalization step used is that explicit malware family names are replaced with a placeholder.

5.1.5 Dataset Splits

For similarity learning, not only preprocessing the data is important, but also the dataset splits. Preventing data leakage is critical in machine learning experiments, since leakage can lead to overly optimistic evaluation results. The split of training, validation, and test datasets impacts what can be measured from an experiment.

The training set for the MLPs was constructed using the 50 largest malware families in the dataset. For each family, 1000 samples were randomly selected in order to create a more balanced training distribution and reduce domination from extremely large families. The rest of the samples are then split into two groups, one of the remaining samples from these families, and one for all samples from families not in the 50 biggest families. The samples from families not in the training set are then split again, 20 families are used for validation and tuning, and the rest form a true hold-out test set. Families containing fewer than 10 samples were excluded from evaluation, since similarity-based evaluation requires a meaningful amount of samples. The same datasets were used for all modalities, meaning that the results are comparable between models. This setup enables evaluation of zero-shot generalization, i.e. the ability of the embedding models to represent malware families not observed during training. Such scenarios are highly relevant in real world malware analysis, where novel or modified malware frequently appears before reliable family

attribution is available.

A challenge with evaluating similarity based malware embeddings is that the composition of the test set affects evaluation metrics. Evaluating on the closed set of only unseen families during training reduces data leakage, but creates a sparse embedding space with fewer competing neighbors, which creates artificially inflated metrics. On the other hand, evaluating the test families in the open set, with seen family samples, better reflects the real world application, but also introduces bias since seen-family clusters are optimized from training, affecting the evaluation metrics also when querying the test samples. Lastly, using the full set of unseen families for both testing and validation is also data leakage, since then one would tune the model towards the data being tested, which is not representative of generalization. To balance these tradeoffs, evaluation was performed on both open and closed test sets. In both cases, only samples from the test split were used as queries. In the open retrieval setting, retrieved neighbors could additionally originate from the validation split, while the closed setting restricted retrieval to test samples only. The tuning is based on the closed set evaluation of the unseen families validation set.

5.1.5.1 Deduplication

As previously discussed, machine learning applied to malware is often more of a data problem than a modeling problem. Generalization often becomes unclear with obfuscation, packers, and ad hoc family labels. All samples in the dataset used in this thesis had a unique binary code, but the difference between two samples could in theory be one byte. To study the effects of the data on model performance, a removal of near-duplicates was performed. Since the raw binaries were not available, only EMBER features, the deduplication step used EMBER features to find near-duplicates. Binaries would have been preferred to use in this step, but since EMBER features are statistical features from the raw binary file, they were deemed sufficient.

The distribution of the cosine similarities can be seen in Figure 5.2. Notice the large peak in intra-family cosine similarity close to one. This indicates that only very small differences exist between these samples, likely originating from packing, recompilation, or minor binary modifications, which is problematic in two ways. Firstly, this will inflate the performance of the embedding space. Without deduplication, highly similar samples may appear across train and evaluation splits, causing overly optimistic performance estimates due to information leakage. Deduplication was performed prior to dataset splitting in order to reduce the risk of this. Secondly, there is little room for the model to learn deeper behavioral similarity, since families are already well separated because of these near-duplicates.

To learn how this was affecting embedding performance, a second dataset was created. This dataset was created by removing one sample from same family pairs with a cosine similarity of more than 0.99 iteratively, until none were left. Although 0.99 cosine similarity threshold may initially appear harsh, in a 672-dimensional feature space it corresponds to extremely similar representations. This led to a dramatic

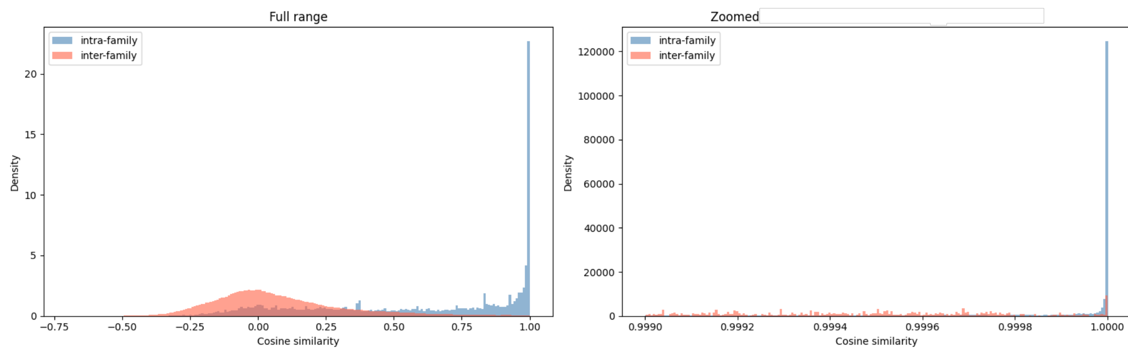


Figure 5.2: Distribution of inter- and intra-family similarity of EMBER features after the scaler was applied. Before scaling, the distributions were just one small peak around zero, and a much larger one close to one. But by preprocessing the data as described above, differences are magnified and a more distributed cosine similarity can be observed, although one can still see a very large peak close to one, indicating that the samples in families do not differ much, leaving very little room for a model to learn.

decrease in samples, keeping only 6% of original samples, and shifting the family distributions. See Figure 5.3 for the fraction of samples kept per family, for a subset of the families in the dataset. Since the deduplicated dataset became that small, creating both an unseen families validation set and a test set was not feasible. Instead this model was tuned by using the unseen samples from seen families, and tested on the unseen families. With a more diverse set of unseen samples from seen families, the idea was that this would still be able to generalize.

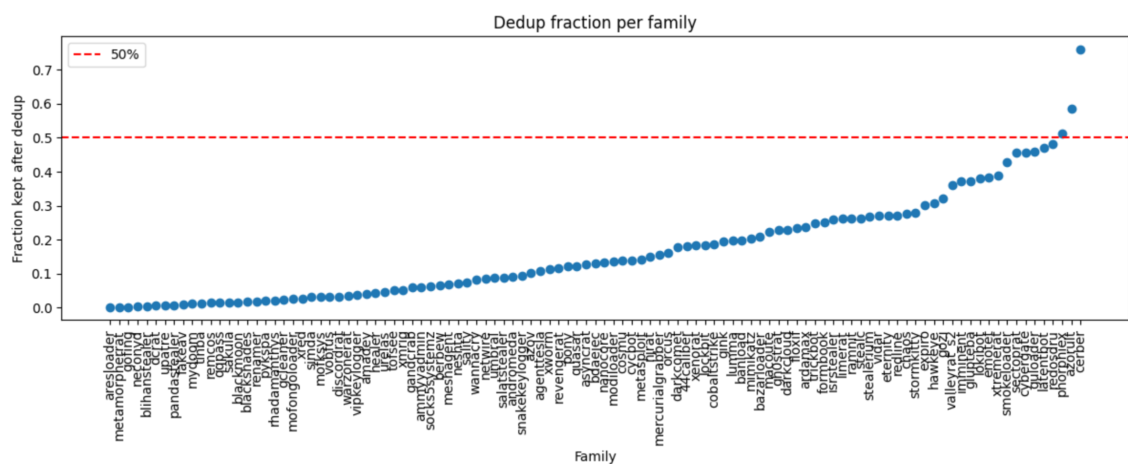


Figure 5.3: The figure shows the fraction of samples kept per malware family after deduplication. The large variation between families indicates that duplicate density differs across the dataset, suggesting that some malware families consist primarily of near-duplicates while others are more diverse.

5.1.5.2 Transformer

Due to computational constraints, the transformer encoder experiments used a slightly different data split for training and evaluation than the Siamese MLP experiments. The initial transformer training dataset contained 5000 report samples from each of the 26 largest malware families. Of these, 90% of the samples were assigned to the training set and 10% to an internal validation set. The internal validation set was used to evaluate seen family performance.

To evaluate zero-shot generalization, a stratified zero-shot validation set of 3000 samples was constructed by sampling 100 samples from the subsequent 30 largest families. These families were not observed during training and were used during experimentation to evaluate the ability of the encoder to cluster unseen malware families. Finally, a non-stratified hold-out test set of 20,276 samples was constructed from all remaining families with more than two available samples. This hold-out set was not used during model training and was reserved for the final evaluation.

While the complete training, internal validation, zero-shot validation, and hold-out datasets were created to support potential future evaluation and further development within Recorded Future, CPU memory and GPU VRAM limitations made it computationally infeasible to use the full datasets for training and evaluation. CPU memory limited the number of preprocessed chunks of samples that could be held in memory, while GPU VRAM constrained the batch size, sequence length, and number of samples that could be processed during training.

To overcome this, the two training stages, MLM and Siamese learning, used different dataset sizes. The MLM step used a uniformly sampled subset of 3000 training report samples and 300 zero-shot validation report samples. The final Siamese fine-tuning used a uniformly sampled subset of 20,000 training report samples for the training set, while the loss was monitored on a reduced, uniformly-sampled subset of 1000 samples from the zero-shot validation set. The MLM used fewer samples, since the model has already learned from large amounts of text and code in its pre-training, leaving little room for improvements. The final hold-out evaluation was performed on a 3000-sample subset of the full hold-out set. This subset was constructed by first ensuring that each included family had at least two samples, after which the remaining samples were uniformly sampled from the hold-out set. Note that the chunking of reports was done after the final dataset split, so that all chunks from a report were in the same dataset.

5.2 Embedding Models

With processed data, embedding models could be trained, to create embeddings for similarity search. In this thesis two different embedding methods were used, namely Siamese MLPs and a transformer. The implementation details are explained below.

5.2.1 Siamese MLP

A multi-layer perceptron (MLP) is a standard feed-forward neural network. The Siamese MLP models were implemented in Python using TensorFlow/Keras. An overview of the architecture for the different modalities can be seen in Table 5.2. These architectures were selected through hyperparameter tuning, see the training section for more details about that. For all modalities, the input feature vector is passed through fully connected layers, to a final embedding space. The output of all hidden layers are batch normalized, an activation function is applied, and finally dropout regularization is used during training. Gaussian Error Linear Unit (GELU) was used as the activation function due to its smooth non-linearity and strong performance in deep learning architectures [65]. After the final embedding layer the embeddings were ℓ_2-normalized.

The hybrid and fusion approaches combine static and dynamic representations in different ways. The hybrid model was trained directly on concatenated static and dynamic feature vectors, allowing the network to learn from both modalities during training. The deduplicated hybrid model was trained on the dataset described earlier where near-duplicates had been removed. The fusion model instead combined embeddings produced by separately trained static and dynamic models. During tuning, Siamese fusion models were evaluated. However, direct concatenation was selected as the final fusion representation used throughout the thesis after evaluation of several trainable fusion architectures during tuning.

Table 5.2: Architecture overview of the final selected Siamese MLP embedding models evaluated in this work. All models used batch normalization and dropout regularization between hidden layers. The activation function used was GELU, and the dropout rate was set to 0.2. The notation under MLP architecture describes the input dimension, hidden layer dimension, and lastly embedding dimension.

Siamese MLP	**Input features**	**MLP architecture**
Static	Static EMBER features	$672 \to 256 \to 64$
Dynamic	Dynamic behavioral features	$622 \to 256 \to 64$
Hybrid	Static + dynamic features	$1294 \to 256 \to 64$
Deduplicated Hybrid	Static + dynamic features	$1294 \to 256 \to 64$
Fuse	Static and dynamic embeddings	Concatenation, $64 + 64 \to 128$

5.2.1.1 Training

Hyperparameter tuning was performed using Optuna for all modalities, and experiment tracking was managed with MLflow. The parameters were selected based primarily on generalization performance on unseen malware families, and secondly on unseen samples from seen families during training. Since the hyperparameter tuning needs scalar evaluation objectives to optimize, Davies-Bouldin index and neighbor purity were used. These metrics were selected because they evaluate embedding quality without being directly dependent on specific hyperparameters such as embedding dimensionality or margin selection. An embedding dimension of 64

was selected since larger embedding spaces did not improve the embedding space while increasing computational cost. In general, shallower architectures with normalized embeddings produced more stable and transferable embedding spaces than deeper alternatives.

Since the primary goal of the thesis was to compare embedding spaces across modalities, the architectures and hyperparameters were kept as similar as possible between models. Because of that, the hyperparameter tuning was under the constraint of maintaining comparable embedding spaces and training conditions, while still being executed for each modality. If no significant improvements of the embedding space were achieved, the shared parameter choices were preferred. In practice, the tuning process therefore focused on optimizing embedding quality and determining stable configurations that generalized across modalities.

See Table 5.3 for the final configuration used. The loss function used is explained more in depth under theory, and is a modified semi-hard triplet loss. For the deduplicated dataset, PK-batching was used to ensure that each batch contained sufficient and family-balanced positive and negative samples for triplet construction, with fewer samples available. For the remaining experiments, standard mini-batches of size 1024 were sampled from a family-balanced training dataset. Family-balanced sampling was used during training to prevent dominance from large malware families and improve learning across smaller families. The model training went for maximum 200 epochs, and early stopping was implemented if the loss did not decrease for 20 epochs. All models were trained using TensorFlow 2.17.1 on a NVIDIA A10G GPU with 24 GB VRAM using CUDA 12.8.

Table 5.3: Training configuration and hyperparameter selection used for the Siamese MLP models. The deduplicated hybrid model used PK-batching due to a limited amount of samples from some families, using $P = 32, K = 16$. Fusion architectures were evaluated during tuning, although the final fusion representation used for evaluation was direct concatenation.

Parameter	**Tested values**	**Final value**
Embedding dimension	16, 32, 64, 128	64
Hidden layers	$[1024, 512, 256, 128, 128]$, $[512, 256, 128]$, $[512, 256]$, $[256, 128]$, $[256]$	$[256]$
Margin	0.1–1.0	0.5
Optimizer	AdamW, SGD	AdamW
Dropout rate	0.0–0.8	0.2
Learning rate	0.001–0.1	0.005
Batch size	256, 512, 1024, PK-batching	1024
Activation function	GELU, Sigmoid	GELU
Loss function	Fixed	Custom Triplet Loss
Embedding normalization	Fixed	True
Weight initialization	Fixed	Xavier/Glorot
Weight decay	0.00001–0.1	0.001

5.2.2 BM25 Reranking

For BM25 reranking, the embeddings for the initial retrieval space were produced by the deduplicated hybrid model. The motivation behind this experiment was to investigate whether lexical reranking of dynamic reports could improve generalization performance, particularly for unseen malware families. However, BM25 retrieval is computationally expensive and not feasible to perform across the full dataset of approximately two million samples at runtime. Therefore, the embedding model was first used to identify a smaller subset of candidate samples for reranking.

For each query sample in a combined set of unseen samples from both families observed and not observed during training, the 100 nearest neighbors in the embedding space were retrieved using cosine similarity. After this, BM25 reranking was applied to the preprocessed dynamic reports for the 100 nearest neighbors, and they were reranked based on BM25 similarity to the query. Purity and hit rate were then computed on the top 10 BM25 matches. These were compared to the purity and hit rate score on the 10 nearest neighbors in the embedding space. The differences for both seen and unseen families were then analyzed.

The reranking stage was motivated by the assumption that embedding similarity captures broader semantic structure, while BM25 can emphasize exact lexical matches in behavioral reports. Combining these approaches could therefore improve retrieval precision for similar malware samples.

5.2.3 Malware Adapted Siamese ModernBERT

The transformer encoder used in this thesis is the ModernBERT-base, which consists of 22 layers and 149 million parameters, trained on 2 trillion tokens of English and code data. Although pre-trained on large amounts of natural language and code, dynamic malware reports differ substantially from conventional natural language domains. Previous research has shown that continued pre-training and domain adaptation on target-domain data can improve performance [66, 67]. Therefore, self-supervised MLM pre-training was first applied, before fine-tuning the model using a supervised Siamese metric learning setup using the triplet loss objective. The combination of these two is results in a domain adapted model.

The purpose of the MLM stage was to adapt the model's contextual token representations to the structure, terminology, and patterns of the malware domain's dynamic reports. These are then used to create sentence and report-level embeddings. The fine-tuning stage then shaped the report-level embedding space such that behaviorally similar samples were mapped closer together than samples from different malware families. Ultimately, the goal was to improve generalization to previously unseen families.

The transformer was trained using PyTorch 2.5.1 on an NVIDIA A10G GPU with 24 GB VRAM using CUDA 12. Hyperparameter-tuning was limited due to the time needed for one run, but parameter choices were chosen by performance on the

validation set when possible. Otherwise they were selected to balance computational feasibility, memory constraints, previous research and training stability.

5.2.3.1 MLM Pre-Training

The actual MLM training pipeline was implemented as a derivative work based on code from the *Sentence Transformers* Python package, which was released under the Apache 2.0 license [58]. It consisted of taking chunks from the MLM training dataset, and feeding them to the MLM training pipeline. The model was then trained and evaluated on its ability to predict masked tokens in the chunks. A full breakdown of the MLM training configuration can be seen in Table 5.4, and the concept of MLM is shown in Figure 4.5.

Table 5.4: Training configurations for MLM pre-training and Siamese metric learning fine-tuning.

Parameter	**MLM pre-training**	**Siamese fine-tuning**
Model	ModernBERT-base	ModernBERT-base MLM-adapted
Loss function	Cross-entropy	SemiHardTripletWithFallbackLoss
Loaded training report samples	3000	20 000
Loaded evaluation report samples	300	1000
Sequence length	512	1026
Training dataset chunk count	227 374	764 604
Evaluation dataset chunk count	17 548	27 745
Train batch size	16	121
Evaluation batch size	16	121
MLM probability	0.15	–
Triplet margin	–	0.4
P in PK-sampling	–	11
K in PK-sampling	–	11
Training epochs	1	2
Learning rate	2×10^{-5}	5×10^{-5}
Optimizer	adamw_torch_fused	adamw_torch_fused
Weight decay	0.01	0.2
Warmup ratio	0.05	0.1
Gradient checkpointing	False	True
Mixed precision	FP16	BF16

5.2.3.2 Siamese Fine-Tuning

For the Siamese fine-tuning, the training was done on a larger dataset, for more epochs, and over a longer sequence length than MLM pre-training. Once again the training pipeline was implemented as a derivative work based on code from the *Sentence Transformers* Python package.

During training, tokenized report chunks were independently encoded using the MLM-adapted ModernBERT in a Siamese setup. Token embeddings were first pooled into chunk-level embeddings according to Equation 4.17. Chunk embeddings belonging to the same malware sample in a batch were then mean pooled into report-level embeddings. The pooled report embeddings were normalized and used for triplet-loss computation according to Equation 4.10.

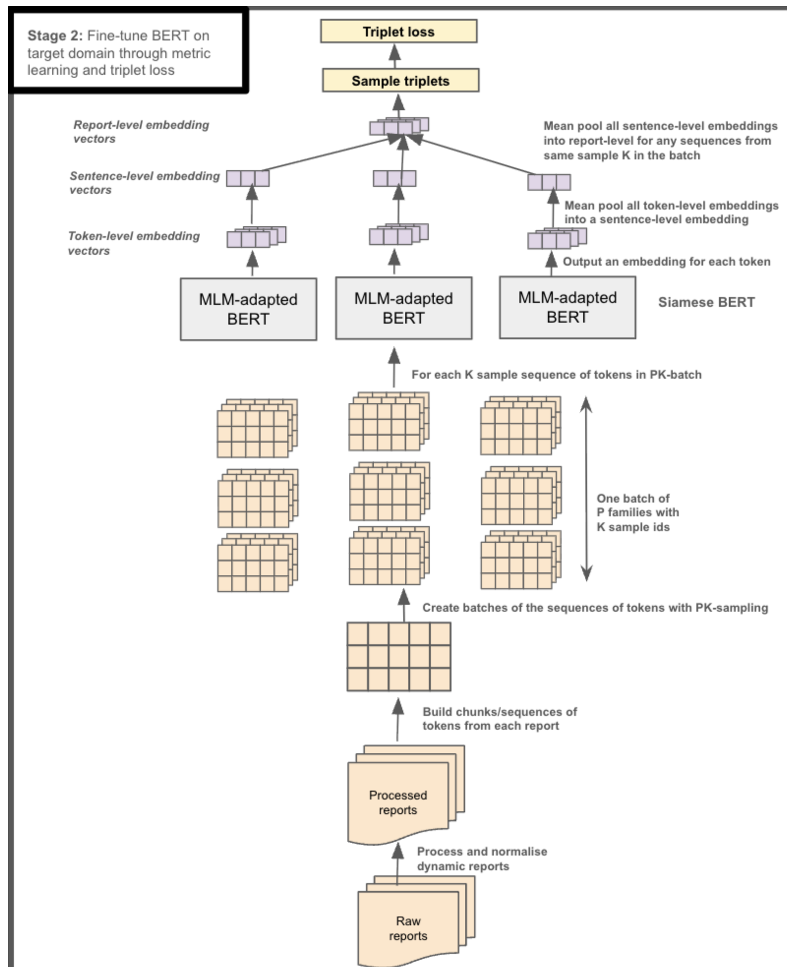


Figure 5.4: Conceptual overview of the domain-adaptation training approach used in this thesis. First, ModernBERT-base is pre-trained on dynamic malware reports using MLM. The resulting encoder is then fine-tuned in a Siamese metric learning setup using triplet loss. Although the figures refer generally to BERT-style encoders, the model used in this thesis is ModernBERT-base.

Using a batch with all chunks from a meaningful amount of reports and families was not feasible. Instead PK-batch sampling was used during training. Within each batch, P malware families were sampled and K chunks were drawn from each family. Chunk selection was stochastic, and multiple chunks originating from the same report could therefore contribute to the pooled report embedding within a batch.

Pooling chunk embeddings into report-level representations better aligned the embedding objective with report-level malware similarity and clustering on broader behavioral similarity, since individual chunks may be noisy, highly specific, or behaviorally incomplete. Across training, the model is then encouraged to learn and understand which semantic patterns remain consistent and discriminative across different report segments and samples within a malware family.

During inference, all chunks belonging to a report were encoded independently and

pooled into a single report-level embedding before retrieval and clustering evaluation. The outline of the Siamese training configuration can be seen in Table 5.4 and a conceptual illustration of this training procedure can be seen in Figure 5.4.

5.3 Evaluation

Embedding quality is inherently difficult to measure, so in order to evaluate the learned embedding spaces, several evaluation metrics for retrieval and clustering were used. Since all embeddings were normalized prior to evaluation, cosine similarity was used as the primary similarity measurement.

To evaluate similarity retrieval performance, nearest-neighbor retrieval was performed in the embedding space using cosine similarity. For each query sample, the $k = 10$ nearest neighbors were retrieved. Two metrics were then evaluated both on unseen samples from known families and on completely unseen malware families in order to measure generalization. Firstly, neighbor purity, the proportion of the ten nearest neighbors belonging to the same malware family as the query sample. Secondly, hit rate, whether at least one of the ten nearest neighbors belonged to the same malware family as the query sample, which can be more meaningful for families with few samples.

To evaluate the global structure of the embedding spaces, clustering metrics were used. The Davies-Bouldin index was used to measure cluster quality by comparing intra-cluster distances with inter-cluster distances. Additionally, angular margins between samples and families were analyzed in order to study how well-separated the learned embedding space was.

Since the learned embedding spaces are high-dimensional, visualization was performed using t-SNE. These visualizations were used to inspect cluster formations, family overlaps, and general embedding structure. The default t-SNE method from scikit-learn was used.

Together, these evaluation methods allow measurements and comparisons of embedding quality, retrieval performance, and clustering. The primary focus of the evaluation was to measure generalization, particularly on unseen malware families, but also on unseen samples from families seen during training.

6
Results

In this chapter, the results produced are presented. The results highlight the inherently difficult nature of learning semantically meaningful malware embeddings that generalize to unseen malware families. In particular, they show that strong performance on unseen samples from seen families does not necessarily imply true zero-shot generalization. Overall, dynamic and hybrid representations provide a stronger basis for similarity learning than only static representations. The findings further suggest that the main limitation is not only model architecture, but also the noisy, imbalanced, and ambiguous nature of malware data itself. The results are interpreted and discussed in the discussions chapter.

6.1 Siamese MLPs

The Siamese MLPs were trained on the same dataset if nothing else is stated, so numerical comparisons between models can be made. But note that the absolute metric values are less important than the relative trends between datasets and training configurations.

6.1.1 Single Modality Embeddings

Results for the static and dynamic Siamese embeddings are shown in Table 6.1. Both modalities achieve strong retrieval results for unseen samples from families seen during training, indicating that the embedding models learn family-specific structures. However, this does not generalize to the unseen families, demonstrating that intra-family generalization does not imply true inter-family generalization.

The static EMBER model already produced strong clustering in the raw feature space prior to training. Siamese training improved retrieval performance on unseen samples from seen families, but did not improve performance on unseen families. This suggests that the static feature space mainly captures statistical and family-specific structure, and not transferable general semantics.

The dynamic model, in contrast, started from a weaker raw feature space, but managed to achieve stronger improvements during training, particularly for unseen family retrieval. This indicates that behavioral information from dynamic reports forms a better basis for semantic similarity learning and zero-shot generalization than static features alone. Notice however, that the learned embedding space is still

Table 6.1: Retrieval performance for static and dynamic embedding models before and after Siamese training. Purity@10 measures the proportion of retrieved neighbors belonging to the same malware family as the query sample, while Hit@10 measures whether at least one retrieved neighbor belongs to the same family. Purity is a per sample average, while hit rate is a per family average, to get both perspectives. The raw evaluation is the feature space, a much higher dimensional space than the embedding space. Since the metrics are dataset dependent, numerical comparisons between the unseen and seen family test cannot be made.

Model	Seen Purity@10	Unseen Purity@10	Unseen Hit@10
Static Raw	81.5%	74.4%	80.2%
Static Trained	93.0%	73.7%	79.1%
Dynamic Raw	70.3%	62.8%	63.1%
Dynamic Trained	90.0%	73.2%	74.2%

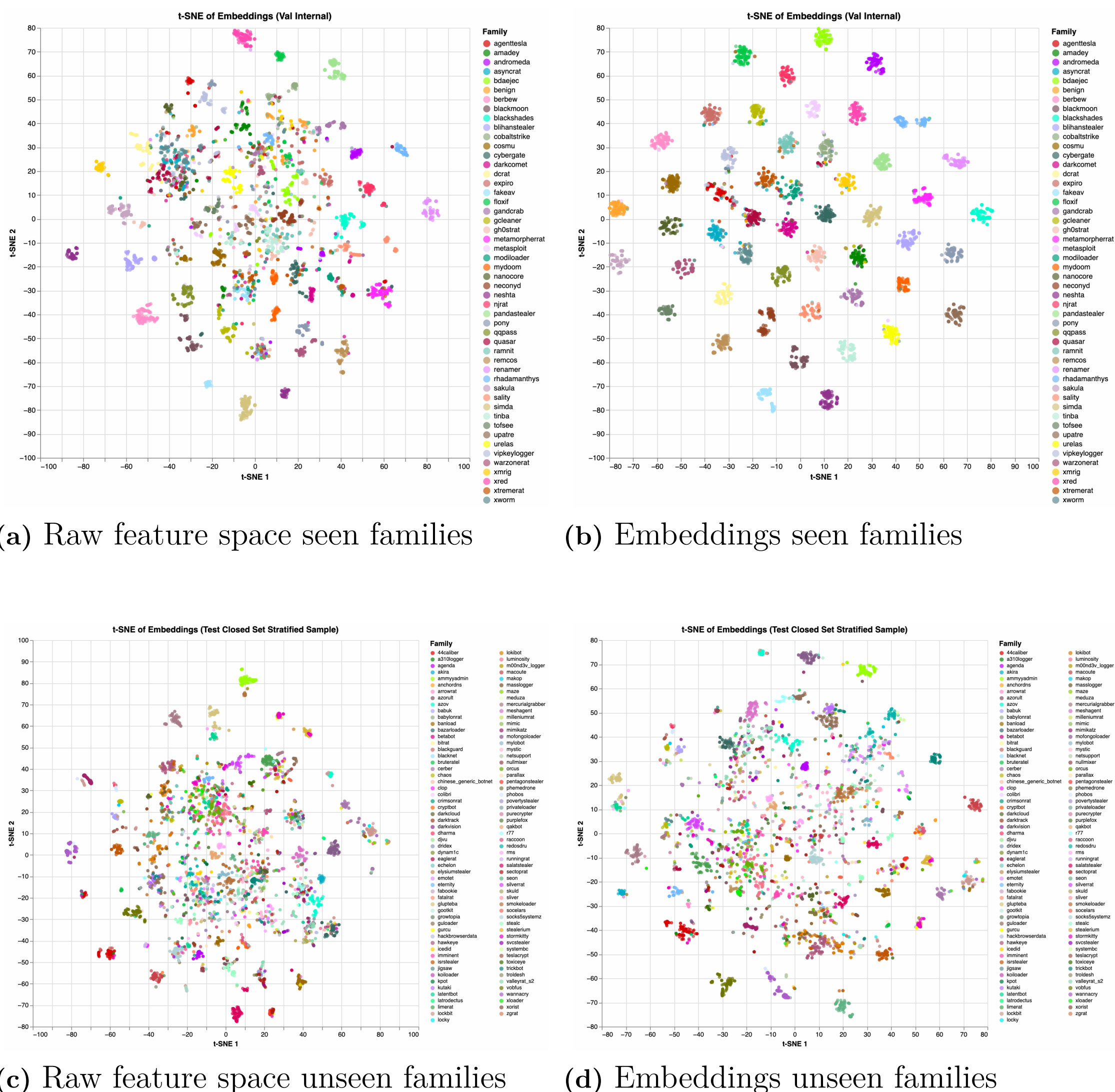
weaker than the raw input static features for unseen families. Overall, generalization to unseen malware families remained difficult for both modalities, and additional t-SNE visualizations of the static and dynamic raw and trained spaces where this behavior can be observed are provided in Appendix D.

6.1.2 Hybrid Embeddings

The learned embedding space can be seen in Figure 6.1. As one can see, the model learns to generalize and separate unseen samples from families trained on, but struggles to do so for families unseen during training. Note that since the unseen family dataset contains more families, and families with fewer samples than the other dataset, the visualization is biased, but the overall pattern is that the well separated family clusters that are in the unseen family embedding space, are already present in the raw feature space. The model struggles to clearly separate families not seen before. The results in Table 6.2 also underlines this. The evaluation metrics improve more for the seen family set than the unseen family set, this is reasonable since they will come from a data distribution similar to the one trained on, but highlights the difficulty of zero-shot generalization.

The fusion model originates from the best static and dynamic embeddings, the ones presented in Table 6.1. The fusion model that performed best on the unseen families validation set was raw concatenation of the embedding vector, all trained models overfitted on the training data. So the embedding space results for the fusion model presented here is in a 128-dimensional embedding space, with just concatenated static and dynamic embeddings.

As one can see the difference between using a fusion model or a hybrid model is almost negligible, with the fusion model generalizing slightly better. Knowing that the fusion model that generalized best was just a concatenation of static and dynamic embeddings, this indicates that the models struggle to learn generalizable correlations between static and dynamic features, only utilizing them separately.



(a) Raw feature space seen families **(b)** Embeddings seen families

(c) Raw feature space unseen families **(d)** Embeddings unseen families

Figure 6.1: t-SNE visualizations of the hybrid representation before and after Siamese training, for both the test set and the set of unseen samples from families seen during training. This is only shown for the hybrid model, not the fusion model, that also showed the same pattern. Notice the well separated clusters in the (b)-plot, those are the unseen samples from families seen during training. The unseen families struggle to form meaningful separated clusters. This shows the difficult task of inter-family generalization, compared to intra-family generalization.

In general however, the hybrid models clearly outperform the single-modality models. The feature space separation from the static representation is combined with the generalizing ability from the dynamic representations, leading to superior evaluation metrics on the unseen samples from seen families, but more importantly, also on the zero-shot dataset. The purity and hit rate show promising results. But as discussed earlier, the numerical values carry little to no meaning, as they are very dataset dependent. In an operational setting that dataset would look different from

the closed test set evaluated here.

Table 6.2: Retrieval performance for hybrid embedding models before and after Siamese training. Purity@10 measures the proportion of retrieved neighbors belonging to the same malware family as the query sample, while Hit@10 measures whether at least one retrieved neighbor belongs to the same family. Purity is a per sample average, while hit rate is a per family average, to get both perspectives. The raw evaluation is the hybrid feature space, a much higher dimensional space than the embedding space. Since the metrics are dataset dependent, numerical comparisons between the unseen and seen family test cannot be made.

Model	Seen Purity@10	Unseen Purity@10	Unseen Hit@10
Hybrid Raw	82.9%	75.6%	81.0%
Hybrid Trained	96.1%	79.7 %	84.4%
Fusion Trained	97.0%	81.6%	86.2%

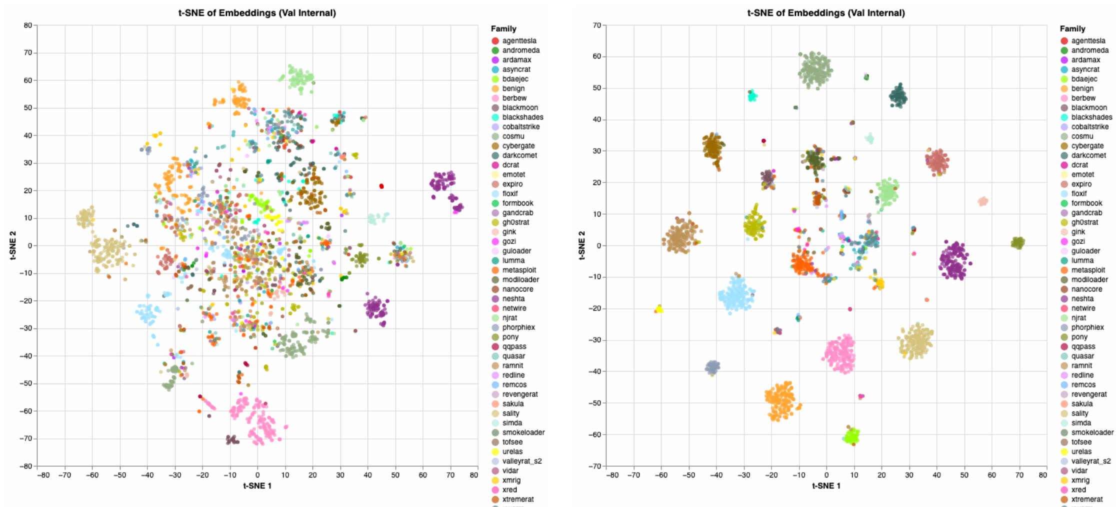
6.1.3 Deduplication and Reranking

Due to the deduplication reducing the amount of valid samples in the dataset by 94%, this model was tuned by using the internal validation dataset consisting of unseen samples from the families trained on. As discussed earlier in this chapter, intra-family generalization does not imply zero-shot generalization, and this can clearly be seen here. With a larger and more diverse dataset, a validation and a test set of unseen families can be created, which would allow the model to be tuned for zero-shot generalization, but it was not possible to create a meaningful split from the few remaining samples from unseen families. The goal with this experiment however, is to show the dataset dependency and study the impact of BM25 reranking, and that can still be studied using this split.

As shown in Figure 6.2 and Table 6.3, the separation in the raw feature space is much weaker than for the original dataset shown in Figure 6.1 and Table 6.2. This is expected, but still interesting, since it shows how much the dataset affects the evaluation metrics. The model however, still learns to map unseen samples from the families trained on well, and the resulting embedding space shows great separation, also for smaller families, which was the goal of the implemented PK-sampling. Regarding generalization, one can clearly see that the model does not cluster the unseen families well, highlighting the difference between intra- and inter-family generalization, even for more diverse datasets.

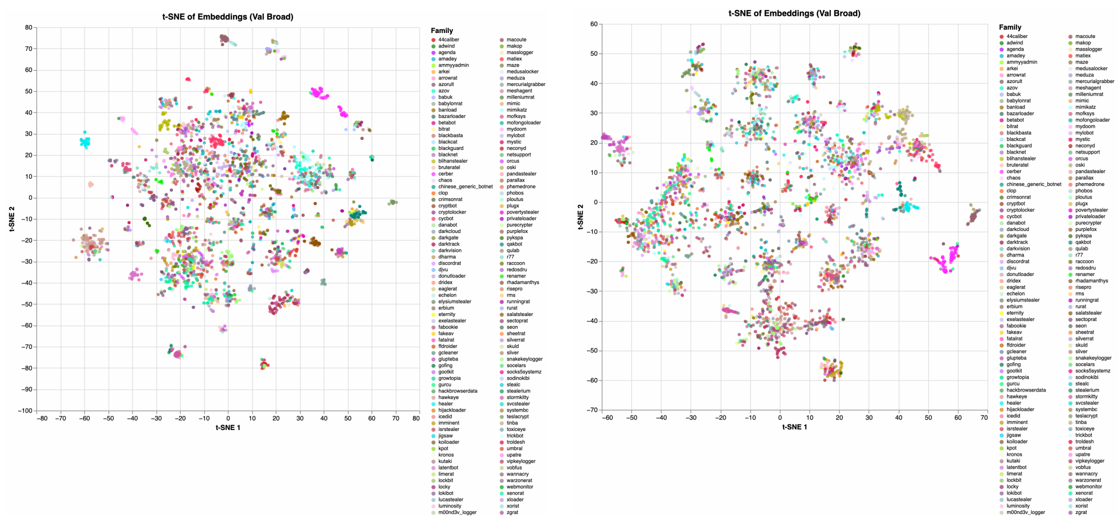
Regarding BM25 reranking, one can see small improvements in the unseen families after reranking, meaning it has an improved zero-shot ability. The lexical reranking can catch overlap in the reports that the embedding model trained on features cannot. The performance of the BM25 is however, embedding dependent, since the BM25 is run on the 100 nearest neighbors in the embedding space. So it is capped from the hit rate at 100. Regarding the unseen samples from the seen families,

BM25 worsens the model. This suggests that the model has already learned the seen families well enough, that lexical reranking has no improvements, the neighborhoods are already well separated on a family level.



(a) Deduplicated raw feature space seen families

(b) Deduplicated embedding space seen families



(c) Deduplicated raw feature space unseen families

(d) Deduplicated embedding space unseen families

Figure 6.2: t-SNE visualizations of the hybrid embedding space on the deduplicated dataset before and after Siamese training, shown for both the hold-out test set and unseen samples from families observed during training. While the model still clusters seen families reasonably well, the deduplicated setting produces more overlap than the non-deduplicated results in Figure 6.1. In contrast, unseen families fail to form clearly separated clusters, highlighting the difficulty of inter-family zero-shot generalization compared to intra-family generalization.

Table 6.3: Retrieval performance for the deduplicated hybrid embedding models before and after Siamese training, and with the added BM25-reranking of the 100 nearest neighbors from the embedding space. Purity@10 measures the proportion of retrieved neighbors belonging to the same malware family as the query sample, while Hit@10 measures whether at least one retrieved neighbor belongs to the same family. Purity is a per sample average, while hit rate is a per family average, to get both perspectives. The raw evaluation is the hybrid feature space, a much higher dimensional space than the embedding space. Since the metrics are dataset dependent, numerical comparisons between the unseen and seen family test cannot be made.

Model	Seen Purity@10	Unseen Purity@10	Unseen Hit@10
Hybrid Raw	60.4%	44.2%	62.4%
Hybrid Trained	79.1%	43.6%	71.4%
With BM25 Reranking	77.4%	47.4%	72.6%

6.2 Malware Adapted Siamese ModernBERT

This section shows the results for the two stages of domain-adaptation of ModernBERT-base. The resulting domain-adapted model is compared against the original ModernBERT-base encoder in order to study how domain adaptation affects clustering quality and zero-shot generalization to previously unseen malware families.

6.2.1 Training and Validation for MLM Pre-training

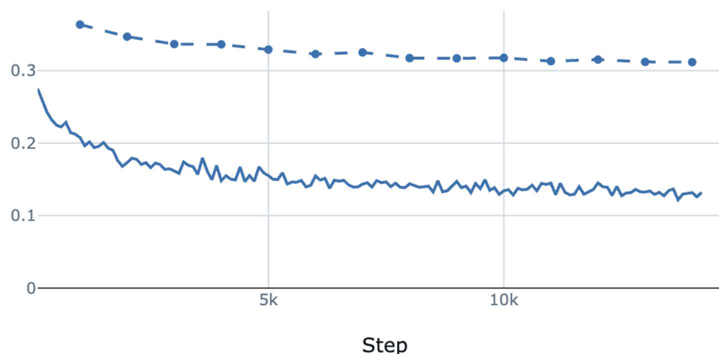


Figure 6.3: Training and evaluation loss curves for the MLM pre-training stage of ModernBERT. Note that the training loss (solid line) is logged throughout training, while the evaluation loss (dashed line) is computed less frequently on the zero-shot validation set.

The MLM pre-training of ModernBERT shows an expected learning behavior, as seen in Figure 6.3. The training loss follows an expected training loss trend, with a decreasing cross-entropy loss. In total, the training loss dropped approximately 54% from its peak to the last step. More interestingly, the same trend can be seen in the evaluation loss in the zero-shot validation set, which decreases by approximately

14%. This indicates that the model improves its ability to predict masked tokens in reports from malware families not observed during training, suggesting that the MLM stage learns contextual representations that generalize beyond the training families.

6.2.2 Training and Validation for Siamese Fine-Tuning

Moving on to the fine-tuned model, the fine-tuning clearly shows an expected loss behavior. As seen in Figure 6.4, the loss curves show decreasing trajectories. The training loss decreases a total of 85%. Once again, the evaluation loss on zero-shot validation set drops, but only by approximately 4.5%.

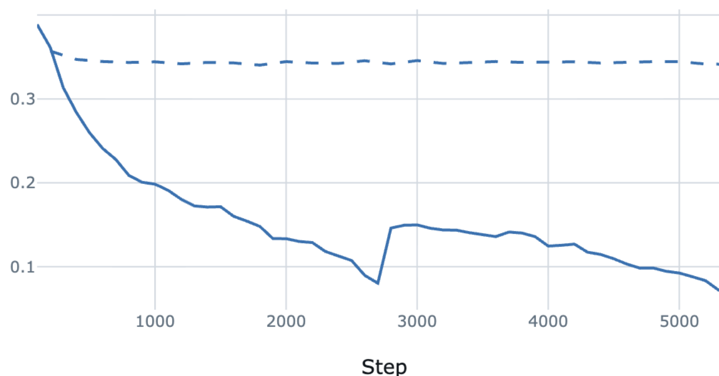


Figure 6.4: Training (solid line) and evaluation (dashed line) loss curves for the Siamese metric learning fine-tuning stage of the domain-adapted ModernBERT model. Note that the training loss is logged throughout training, while the evaluation loss is computed less frequently on the zero-shot validation set.

6.2.3 Evaluation of Embeddings

A good baseline to use as reference for the performance of the final domain adapted ModernBERT, is the original ModernBERT model. Using the original model as the reference point allows us to see the effects of the domain adaptation. As shown in Figure 6.5, the original ModernBERT shows poor separation for many families over all datasets. Note however, the existence of already formed clusters for certain families in all t-SNE plots. This suggests that some malware families contain similar dynamic reports that even the non-domain adapted ModernBERT model can separate their report embeddings to some extent.

The effects of the training become especially clear when looking at the t-SNE plots for the different datasets in Figure 6.6. The training and internal validation datasets' t-SNEs for the domain adapted ModernBERT, as seen in Figures 6.6a and 6.6b, clearly show distinct clusters for all the families trained on, compared to the base model shown in Figures 6.5a and 6.5b. For the zero-shot validation set, the domain adapted model in Figure 6.6c shows some new cluster formations and clearer separation relative to the base model in 6.5c. Finally, for the hold-out it is not as clear, since it contains many small families, but similar results can be seen between

Figures 6.6d and 6.5d.

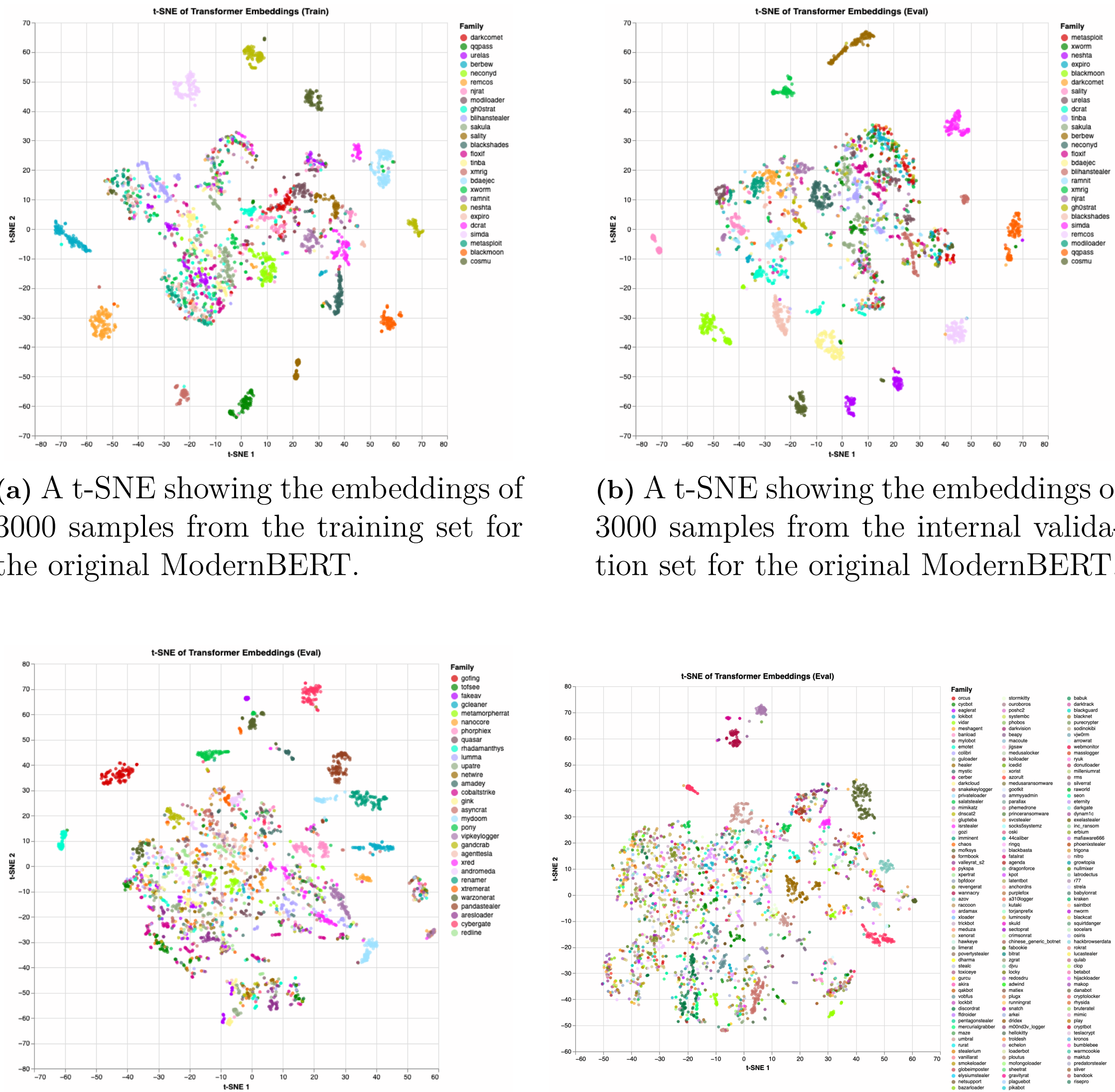
To overcome the difficulties in visual analysis, another view of the effects of the domain adaptation can be found in Table 6.4, which gives a numeric overview of the changes. This is especially important for the zero-shot and hold-out sets, where the t-SNEs are harder to visually analyze. As shown in the table, the domain adapted ModernBERT improves for all values. The train and internal validation sets improve to basically perfect purity whilst the Davies-Bouldin index drops. The zero-shot validation set shows improvement as well for both metrics, indicating that the model is able to improve on unseen families. Finally, the hold-out set also shows improvements across all metrics. In particular, the most interesting result is that the hit rate increases from 33% to 46%. The improvements suggests that the domain adapted model is able to, to some extent, improve zero-shot family clustering in the hold-out set with smaller families.

Table 6.4: Evaluation metrics for Purity@10, Davies-Bouldin index, and Hit@10 for the baseline ModernBERT model and the domain adapted ModernBERT model across the four dataset splits. Purity@10 measures the proportion of retrieved neighbors belonging to the same malware family as the query sample, while Hit@10 measures whether at least one retrieved neighbor belongs to the same family. Purity is a per sample average, while hit rate is a per family average, to get both perspectives. Note that hit-rate is only reported for the hold-out set, since the metric is most meaningful for families with few samples. For families with more samples, the interesting metric is purity.

Model	**Metric**	**Training**	**Internal val.**	**Zero-shot val.**	**Hold-out**
ModernBERT	Purity@10	72.8%	74.8%	62.1%	44.6%
	Davies-Bouldin	4.0	4.3	4.7	4.2
	Hit@10	-	-	-	33.1%
Domain Adapted ModernBERT	Purity@10	97.1%	96.3%	77.9%	57.3%
	Davies-Bouldin	0.71	0.78	2.9	2.5
	Hit@10	-	-	-	45.8%

Note that these results should interpreted cautiously. First, due to the computational limitations described in the methods chapter, the metrics are computed on relatively small subsets of the full datasets. In sparse embedding spaces, this may artificially increase clustering metrics, since fewer samples need to be separated. Second, as seen in the t-SNE visualizations of the baseline ModernBERT, some dynamic reports appear to be near-duplicate samples that already embed close together before any domain adaptation.

In a real world setting, the embedding space would contain a much larger and more diverse set of samples; meaning the absolute metric values are not indicative of large-scale performance. With that being said, the relative trend still describes a meaningful finding: the domain-adapted ModernBERT is consistently better than the non-adapted, original model across all datasets at zero-shot clustering and generalization.



(a) A t-SNE showing the embeddings of 3000 samples from the training set for the original ModernBERT.

(b) A t-SNE showing the embeddings of 3000 samples from the internal validation set for the original ModernBERT.

(c) A t-SNE showing the embeddings of 3000 samples from the zero-shot validation set for the original ModernBERT.

(d) A t-SNE showing the embeddings of 3000 samples from the hold-out set for the original ModernBERT

Figure 6.5: *t*-SNE visualisations of report-level embeddings produced by the original ModernBERT model as reference, using a sequence length of 1026 for each chunk. Each subplot shows 3000 samples from one dataset split: the training set, the internal validation set, the zero-shot validation set, and the hold-out set.

(a) A t-SNE showing the embeddings of 3000 samples from the training set for the domain-adapted ModernBERT.

(b) A t-SNE showing the embeddings of 3000 samples from the internal validation set for the domain-adapted ModernBERT.

(c) A t-SNE showing the embeddings of 3000 samples from the zero-shot validation set for the domain-adapted ModernBERT.

(d) A t-SNE showing the embeddings of 3000 samples from the hold-out set for the domain-adapted ModernBERT.

Figure 6.6: *t*-SNE visualizations of report-level embeddings produced by the domain-adapted ModernBERT model using a sequence length of 1026. Each subplot shows 3000 samples from one dataset split: the training set, the internal validation set, the zero-shot validation set, and the hold-out set.

7
Discussion

This chapter interprets and discusses the results presented in the previous chapter, focusing on zero-shot generalization and limitations of the evaluation. The findings indicate that while meaningful embedding spaces can be learned, especially from hybrid representations, true zero-shot generalization remains challenging. The results further suggest that malware similarity analysis challenges are not only related to model architecture, but also to dataset structure, weak family labels, and obfuscation of malware samples.

7.1 Static vs. Dynamic Malware Representations

An interesting result, shown in Table 6.1, is that the static EMBER representation of malware better separates the malware families in the input feature space than the manually extracted dynamic features, but that the Siamese MLP can learn zero-shot generalizing patterns from the dynamic representations, and not from the static representation. This is expected, since the EMBER features are PE-file statistics mostly and the family classification rules are mostly based on the binary files, while the dynamic features show runtime behavior. But interestingly, the zero-shot metrics are better for the raw static input feature space, than for the trained dynamic model, showing that the initial separation is strong using EMBER features, and that the dynamic model struggles to find an equally strong separation of families even when trained. Overall, the static feature space provides stronger family separation than the dynamic representation, although this separation appears less transferable to unseen malware families.

Other static representations exist as described earlier, and have not been explored in this thesis. They might be able to learn zero-shot generalization, so no general conclusions can be made about static representations, but the results suggest that EMBER features alone are limited for zero-shot similarity learning under family label objectives. Instead, dynamic representations are more promising. The features manually extracted from dynamic reports used in this thesis are not necessarily the best possible representation, and a more refined dynamic representation could potentially improve similarity learning.

EMBER features are problematic in more than one sense for similarity learning. Firstly they are statistical features, like byte counts and entropy, which form strong separation and can be used to identify known distributions, but clearly struggle for

unseen distributions. This is reasonable as such features do not track behavior, and thus struggle to generalize to unseen families, since what the model learns from the representation is not behavior, it is binary file fingerprints. Another issue with using EMBER features is that many malware files are packed, meaning that they are compressed high entropy files, that need to be unpacked before running. Extracting statistical features from those does not give behavioral information, it gives packer artifacts, which are not generalizable. Overall the static EMBER features give the model access to family fingerprints, instead of family behavior, which struggles to generalize to unseen families, but very strongly identifies similar samples from seen families.

The dynamic representation however, survives obfuscation and packing much better. They do not represent the binary code, rather the behavior, which is a more suitable representation for the task. If two binaries differ much, but behave similarly, that is already encoded in the dynamic representation but not in the static representation. And in the end, that is what the preferred outcome is, to map behaviorally similar samples together.

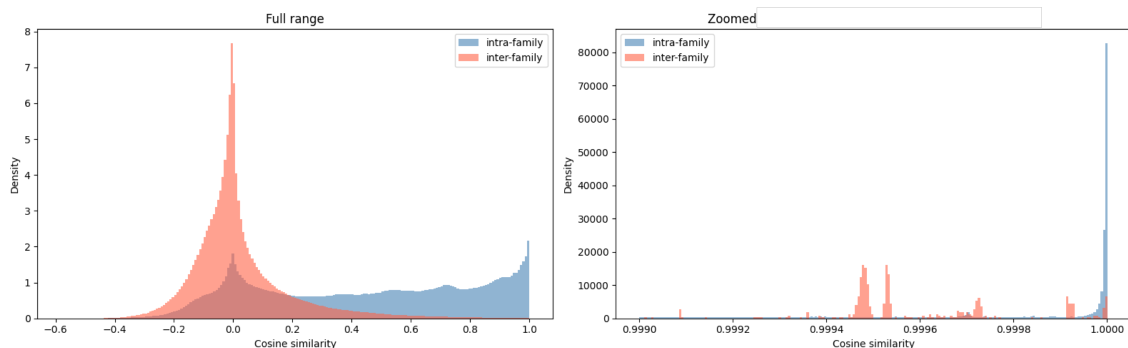


Figure 7.1: Distribution of inter- and intra-family cosine similarity of manually extracted dynamic feature vectors after preprocessing and scaling. Compared to the static EMBER representation in Figure 5.2, the dynamic representation exhibits broader intra-family variation and greater overlap between malware families.

Figure 7.1 helps explain why the dynamic representation generalizes better than the static EMBER representation. The dynamic features exhibit broader intra-family variation and greater overlap between malware families, compared to the EMBER features shown in Figure 5.2. This indicates that the representation captures behavioral structure rather than only sparse statistical fingerprints. In contrast, the static EMBER representation contains malware families with very small internal variance that are often close to orthogonal to each other in the feature space. Consequently, the static representation becomes highly separable but less transferable across unseen families. This suggests that highly separable embedding spaces are not necessarily more semantically meaningful than weaker but more transferable representations.

7.2 Hybrid Representations and Learning

By combining the initial strong separation in the EMBER feature space, with the generalizing ability of the dynamic models, stronger embedding spaces are achieved. Interestingly, the fusion net cannot find a better representation generalizing better than just concatenating the static and dynamic vectors. This may indicate that the separately learned embedding spaces already capture most of the information available under the family-label objective, making additional fusion learning impossible without overfitting. Also the retrieval performance comparing a model originally trained on a hybrid representation with the fusion model does not differ much. This indicates that for the hybrid models, the functionality is highly divided, the static representations form better clusters in the input feature space, while the dynamic representation enables generalization.

But hybrid models clearly outperform the single modality embedding models, indicating that this is a better alternative for effective similarity learning. This is an expected result as more information is being given to the model, but it clearly shows that static and dynamic representations contain different information, and one is not sufficient to get the full picture of a malware. The results suggest that the static and dynamic representations encode different and complementary information. The static EMBER representation provide strong family separation through statistical features from the binary file, while the dynamic representations provide more transferable behavioral structure. Combining them therefore improves both local family clustering and zero-shot performance.

7.3 Transformers and Their Ability to Generalize

As shown in the results, the domain adapted ModernBERT shows promising potential in zero-shot clustering and generalization. This is expected, since the manually extracted dynamic features, which are derived from the same reports, also show some ability to improve the separation unseen in family embeddings. However, the benefit of using a domain adapted ModernBERT on these reports, is that it does not require the same level of manual feature engineering to identify the most semantic meaningful characteristics. This is because, a priori, it is easier to remove or mask clearly superfluous or obfuscated information than to extract the most discriminative and semantically meaningful features. Instead, the attention mechanism of a transformer, combined with the 2-stage training process of domain adaptation, allows the model to identify those characteristics by itself and create report-level embeddings encoding this information.

However, as stated, the absolute values in the results should be interpreted with caution. This is because the sample size used is simply too small to give a definite answer to how this method would work in an operational setting with a much larger and more diverse set of samples. Moreover, despite the improvements in for example the hold-out set, the efficacy is most likely too low to be applicable for downstream

tasks. More concisely, even if the model improved the hit rate to 46 % for example, it stills means that more than half of the samples have no other samples from the same family in its 10 closest neighbors.

More limitations of this approach is the extensive computational resources required to create a complete pipeline. First, it requires a large-scale generation of dynamic reports in a sandbox environment. Second, these reports must be preprocessed into a suitable structure for the encoder. Third, training the encoder on a sufficiently large dataset requires a distributed training setup on several GPUs, especially if larger encoder models or longer sequence lengths are used. Lastly, the inference stage is also computationally expensive. Each report must be split into chunks, each chunk must be encoded into a sentence embedding, and the resulting chunk embeddings must then be pooled into a single report-level embedding. Even with the reduced training used in this thesis, only 20 000 report samples produced 764 604 chunks of 1026 tokens each. The demonstrates the how quickly the computational demand can grow when using this approach.

7.4 Data Quality and Malware Family Ambiguity

As indicated in the results, malware research often turns into a data problem, instead of a model problem. The quality of the data is difficult to determine, especially when it comes to near-duplicates. In the operational environment, a model would see a lot of these near-duplicates and needs to be able to classify them. But that is not the difficult part of malware similarity research, that is finding general patterns that can make diverse family samples together. Therefore, having many near-duplicates in the data, inflates the evaluation metrics, and makes the true zero-shot generalization difficult to track. In this thesis this can be seen in the performance drop when using the deduplicated dataset. By getting rid of the near-duplicates, all evaluation metrics drop dramatically. This shows how dataset dependent malware similarity learning is, and why comparisons are difficult if the same data is not being used. Looking back at Figures 5.2 and 5.3, one can also see why performance on unseen samples from seen families is not a good measure of generalization, when the internal variation is negligible for many families.

Figure 7.2 further illustrates the EMBER representation after preprocessing the data. While scaling improves the similarity structure compared to the raw feature space, many families remain orthogonal to each other, while being very similar internally. This orthogonality likely originates from sparse statistical feature activations, where families activate unique combinations of the EMBER feature space. This indicates that similarity in EMBER is driven by sparse family feature fingerprints, rather than transferable behavioral semantics. At the same time, some family clusters can be seen, suggesting that certain families share structural characteristics, but it is not clear if that is driven by using the same packer or other obfuscation methods, or actual code similarity. Overall, the overlap between families remains weak, which helps explain why the static representations achieve strong family sep-

aration while struggling to generalize to unseen families.

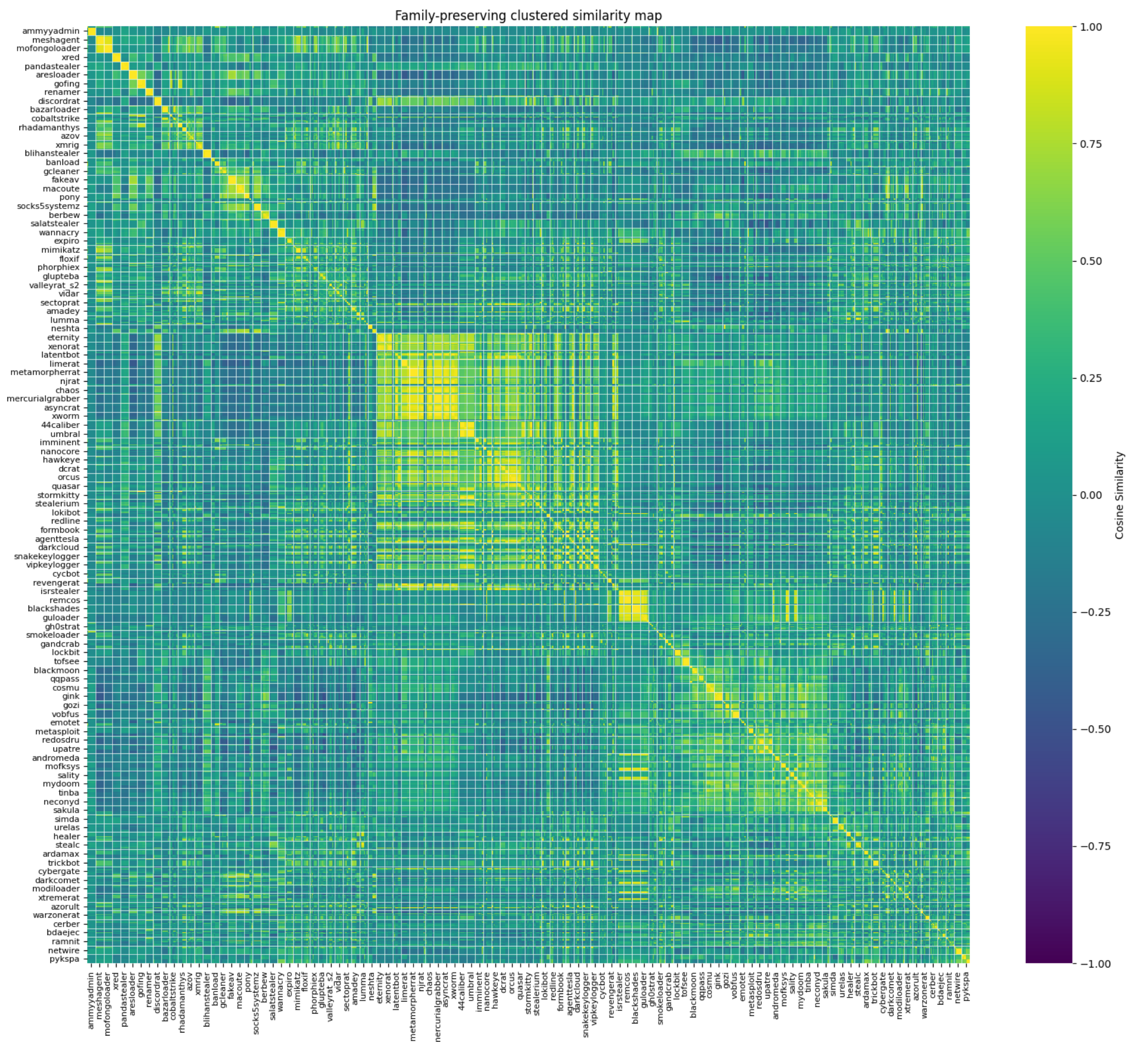


Figure 7.2: Family grouped cosine similarity in the scaled EMBER feature space, where each family is represented by 50 samples. Although some malware families form local similarity clusters, many families remain close to orthogonal in the feature space, indicating sparse statistical separation rather than globally shared semantic structure, aligning with Figure 5.2.

Another problem with similarity learning applied to malware, is that when using family labels as the target, you depend on them being behaviorally separating labels. That is not entirely true, as they are created by AV-researchers ad hoc, and are not a perfect target for similarity learning. Samples belonging to the same family may differ substantially in functionality and behavior, while samples from different families may share significant behavioral overlap or code reuse. For example, Remote Access Trojans (RATs), a bigger class of malware that contains many families, may have family overlap, something that the family separating triplet loss function does not align with. The models enforce family separation even in cases where malware families may share substantial behavioral overlap or code reuse. However, defining

semantically meaningful similarity objectives for malware remains difficult. One needs to take this into account when evaluating results, the ground truth labels do not guarantee semantically meaningful (dis-)similarity, only an ad hoc family label. This makes evaluation difficult, since high retrieval performance with respect to family labels does not necessarily imply that the embedding space reflects meaningful behavioral relationships. Consequently, the learned embedding spaces may capture dataset fingerprints and labeling conventions rather than true semantic malware similarity.

7.5 Retrieval Refinement with BM25 Reranking

As the results above show, lexical reranking can improve the retrieval from the learned embedding spaces, especially for zero-shot retrieval. Interestingly, it does not improve the retrieval when querying seen samples from unseen families. The experiment was conducted on hybrid embeddings, indicating that the information used for the lexical reranking step is already encoded in the embedding space to some extent. The lexical reranking is a more computationally expensive method than the pure embedding based retrieval, and is therefore only conducted on the 100 nearest neighbors of the query. This limits the reranking, if no samples from the same family are present in the 100 nearest neighbors, the reranking cannot find a correct match.

In an operational setting, this is a promising method to match two unseen family samples together, in true zero-shot retrieval. A multi stage method could potentially work well for this, since running BM25 on large datasets scales badly. But a stage one embedding method determining if a malware sample belongs to a family seen during training or not, and if not, BM25 on a representative set of known and unknown samples to find best matches could be performed. This has not been explored in this thesis, but could potentially work well for true zero-shot generalization.

7.6 The Challenge of Evaluating Zero-Shot Generalization on Malware

A more general observation that can be inferred from the findings in this thesis, is the difficulty of measuring similarity learning when it comes to zero-shot generalization. Whilst it can be achieved to some extent, all performance metrics are dataset dependent, where data leakage with for example near-duplicates is difficult to avoid, if one is not to evaluate on a small subset of data. Consequently, the absolute metric values are less important than the relative trends observed between datasets, modalities, and preprocessing strategies. Moreover, more modern approaches like transformer encoders require an extensive data pipeline which might be unfeasible for the scale of the problem. Interestingly, if the objective of this thesis had instead been malware classification, as is common in malware research, the models would have achieved very strong performance when evaluated on unseen samples

from malware families observed during training. This highlights both the diversity of malware data and the difficulty of comparing results across different datasets and evaluation methodologies. A lot of research has been and is being conducted, but the comparability of most studies is limited.

Interestingly, as one can see in the t-SNE plots for the Siamese MLPs, the models can generalize to unseen goodware when trained on other benign files. Benign files should have more internal diversity than a malware family, and this shows that although not the goal of the thesis, the models learned a well defined boundary between goodware and malware. Since the goodware is taken from a specific, different dataset however, this cannot be claimed to be valid for all goodware.

7.7 Operational Implications

In an operational environment, malware analysis is challenging. Threat actors modify their malware all the time, and by using AI, they can do it on a new scale. Cybersecurity actors try to train machine learning models to detect and classify malware, and the malicious actors try to avoid detection. A classification model working today will at some point stop working. Therefore, zero-shot generalization is needed, but to perform that at scale is difficult. But what this thesis shows is that if one wants a model able to generalize to unseen families, focus should be on the data. A representative and large dataset is necessary for this purpose, and still the model will struggle with out of distribution data. Obtaining dynamic representations of malware at an operational scale is more expensive than the static binary analysis, but the thesis shows that dynamic representations are necessary for zero-shot generalization.

A similarity model used in an operational setting would need to be retrained often, on updated representative datasets, in order to have the conditions that enable similarity search. Cybersecurity will always have to deal with emerging families and distribution shifts, and this thesis shows that without representative data, zero-shot generalization is very difficult.

7.8 Ethical Considerations

This thesis focuses on cybersecurity, but as for all malware research, the insights can also potentially help attackers. One needs to take this into account, but the thesis is a defensive analysis of malware behavior and similarity, looking at the struggles of zero-shot generalization. It does not focus on how to avoid the system, just how the system works. But a malicious actor can still learn from this, learn how and why similarity models are failing, and this dual-use should not be forgotten when publishing malware research.

A subject that has been focused a lot on is the lack of reproducible and comparable experiments in the field of malware analysis, and to then not publish the dataset

used might seem strange. This is not possible due to legal and operational constraints, but the experiments are reproducible since the feature extraction methods can be applied to any PE file and sandbox report. The need however, of a public, up to date dataset, is evident in the field of malware analysis, and would greatly improve the field by making results comparable and experiments reproducible.

Another data issue shown here is that the embedding results are heavily data dependent. One cannot expect the same results on a different dataset and this can give a false sense of generalization in automated cybersecurity systems. This risks an overestimation of real world performance for ML systems applied to the field.

8
Conclusion

The thesis investigated malware similarity embeddings using static, dynamic, and hybrid representations through Siamese metric learning. The results show that generalizing to unseen samples from families observed during training does not necessarily imply true zero-shot generalization. Or in other words, all malware looks similar until it does not. While metric learning can successfully capture similarity among families represented during training, the transition to previously unseen malware families remains a fundamentally difficult challenge. This highlights that apparent similarity in malware does not necessarily correspond to transferable semantic structure.

Static EMBER features mostly encode sparse family fingerprints rather than transferable behavioral semantics. Dynamic representations, regardless of the model used, provided better transferability and generalization than static EMBER features, while both of the hybrid representations achieved the strongest overall performance by combining statistical file information and behavioral information. However, the experiments also showed that malware similarity learning is heavily constrained by the quality and structure of the data. Near-duplicate samples, sparse family specific fingerprints, and ambiguous malware family labels affect both learned embedding spaces and evaluation metrics.

The findings suggest that the main limitation of malware similarity learning is not necessarily the model architecture, but rather datasets, family labels, and evaluation methodology. Future work should therefore not only focus on more advanced embedding models, but also on improved datasets, behaviorally meaningful similarity objectives, and more robust zero-shot evaluation methods.

9

Future Work

The biggest issue with this domain is the lack of reproducibility between various research papers and experiments due to the lack of open-source datasets, especially for dynamic or hybrid modalities of malware. The EMBER dataset partly offers that, but is limited to static representations and, as this thesis has shown, is unsuitable for similarity learning tasks. Moreover, as stated in the discussion, the concept of generalization within malware similarity is dataset dependent. Therefore, it is important to define different malware datasets that are representative of the current landscape of the domain and the task to be achieved. For example, curating a hold-out set that properly captures the type of generalization a model should reach in order to be useful in a real world setting. Another generalization direction for future work, for which defining a benchmark dataset is important, is evaluating robustness to temporal distribution shifts and malware evolution over time. Since malware behavior and obfuscation techniques continuously evolve, embedding spaces should ideally remain stable under changing threat landscapes.

Moving on to the findings and work of this thesis specifically, a valuable direction for future work would be to do deeper ablation studies on the static and various dynamic representations used in this thesis. For the MLP-based models, this could involve analyzing which feature groups contribute most to semantic similarity and zero-shot generalization. For the Transformer and reranking methods, it could involve studying which parts of the dynamic reports contain the most important behavioral information. This is especially important given the computational cost of these methods. If the reports could be shortened while preserving the most semantically meaningful information, such approaches would become more feasible to scale to real world datasets.

This is especially relevant for the manually extracted dynamic features. Since we do not have deep domain-expertise in the malware domain, it remains unclear whether the selected features capture the most semantically meaningful information, whether only a smaller subset is required, or whether other features would be more robust to distribution shifts. In addition to this, other static representations than EMBER features should be explored for the malware similarity area. They can potentially form the basis for more meaningful and transferable embeddings.

Finally, while this thesis explored the use of combining different modalities to try to achieve better generalization, no deeper exploration has been done on how models can be combined in a multi-stage decision process to achieve different tasks. Some

initial experiments were performed using BM25-based re-ranking, but this direction requires further research.

More generally, even though malware-family definitions can be relatively ad hoc, expert analysts follow an implicit decision process when determining whether a sample belongs to an existing family or represents the beginning of a new one. Potentially, that process can be recreated with machine learning models. An idea could, for example, be to use static representations to filter out samples from families trained on. Then, for the remaining samples, a combination of transformer encoders, manually extracted dynamic features, and unsupervised methods could be used to support a weighted similarity or attribution decision on unseen families.

This type of multi-stage approach may be more realistic than relying on a single monolithic model. Given the described issues with the underlying data for all modalities, it is unlikely that one model alone can fully solve the problem of malware similarity. Overall, however, the focus should first be on creating well structured datasets that these models can learn from.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[2] Daniel Voigt Godoy. Bert embeddings 01, 2021. File: BERT embeddings 01.png. Licensed under CC BY 4.0.

[3] Daniel Voigt Godoy. Bert masked language modelling task, 2021. File: BERT masked language modelling task.png. Licensed under CC BY 4.0.

[4] Hyrum S. Anderson and Phil Roth. EMBER: an open dataset for training static PE malware machine learning models. *CoRR*, abs/1804.04637, 2018.

[5] Mesut Guven. Dynamic malware analysis using a sandbox environment, network traffic logs, and artificial intelligence. *International Journal of Computational and Experimental Science and Engineering*, 10, 09 2024.

[6] Dutch Ministry of Defence. Public annual report 2025 military intelligence and security service. 2026.

[7] Willem Buys Kilian Hayat Giulia Moschetta, Ellie Winslow. Global cybersecurity outlook 2026. 2026.

[8] August Klynne and Malte Åqvist. Don't judge a malware by its binary: Similarity-based embeddings for efficient malware analysis. Master's thesis, Chalmers University of Technology, 2025.

[9] Aziz Mohaisen and Omar Alrawi. Av-meter: An evaluation of antivirus scans and labels. In Sven Dietrich, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 112–131, Cham, 2014. Springer International Publishing.

[10] Yao Saint Yen, Zhe Wei Chen, Ying Ren Guo, and Meng Chang Chen. Integration of static and dynamic analysis for malware family classification with composite neural network. *CoRR*, abs/1912.11249, 2019.

[11] Dmitrijs Trizna. Quo vadis: Hybrid machine learning meta-model based on contextual and behavioral malware representations. In *Proceedings of the 15th ACM Workshop on Artificial Intelligence and Security*, CCS '22, page 127–136. ACM, November 2022.

[12] Microsoft. Microsoft digital defense report 2024. `https://cdn-dynmedia-1.microsoft.com/is/content/microsoftcorp/microsoft/final/en-us/microsoft-brand/documents/Microsoft%20Digital%20Defense%20Report%202024%20%281%29.pdf`, 2024. Accessed: 2026-05-08.

[13] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification, 2016.

[14] Gopinath M. and Sibi Chakkaravarthy Sethuraman. A comprehensive survey on deep learning based malware detection techniques. *Computer Science Review*, 47:100529, 2023.

[15] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers Security*, 81:123–147, 2019.

[16] Saranya Chandran, Sreelakshmi R Syam, Sriram Sankaran, Tulika Pandey, and Krishnashree Achuthan. From static to ai-driven detection: A comprehensive review of obfuscated malware techniques. *IEEE Access*, PP:1–1, 01 2025.

[17] Dragos Georgian Corlatescu, Alexandru Dinu, Mihaela Gaman, and Paul Sumedrea. Embersim: A large-scale databank for boosting similarity search in malware analysis, 2023.

[18] Ömer Aslan Aslan and Refik Samet. A comprehensive review on malware detection approaches. *IEEE Access*, 8:6249–6271, 2020.

[19] Eugene Spafford. The internet worm program: An analysis. *ACM SIGCOMM Computer Communication Review*, 19, 03 2000.

[20] Matthew Schultz, Eleazar Eskin, F. Zadok, and Salvatore Stolfo. Data mining methods for detection of new malicious executables. pages 38–49, 02 2001.

[21] J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, December 2006.

[22] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. Malware detection by eating a whole exe, 2017.

[23] Sang Ni, Quan Qian, and Rui Zhang. Malware identification using visualization images and deep learning. *Computers Security*, 77:871–885, 2018.

[24] Maksim E. Eren, Manish Bhattarai, Kim Rasmussen, Boian S. Alexandrov, and Charles Nicholas. Malwaredna: Simultaneous classification of malware, malware families, and novel malware, 2023.

[25] Maksim E. Eren, Ryan Barron, Manish Bhattarai, Selma Wanna, Nicholas Solovyev, Kim Rasmussen, Boian S. Alexandrov, and Charles Nicholas. Catch'em all: Classification of rare, prominent, and novel malware families, 2024.

[26] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2), March 2008.

[27] Ulrich Bayer, Paolo Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. Scalable, behavior-based malware clustering. 01 2009.

[28] Zhaoqi Zhang, Panpan Qi, and Wei Wang. Dynamic malware analysis with feature engineering and feature learning, 2020.

[29] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Comput. Surv.*, 52(5), September 2019.

[30] Pradip Kunwar, Kshitiz Aryal, Maanak Gupta, Mahmoud Abdelsalam, and Elisa Bertino. Sok: Leveraging transformers for malware analysis, 2025.

[31] Dmitrijs Trizna, Luca Demetrio, Battista Biggio, and Fabio Roli. Nebula: Self-attention for dynamic malware analysis. *IEEE Transactions on Information Forensics and Security*, 19:6155–6167, 2024.

[32] Tony Quertier, Benjamin Marais, Grégoire Barrué, Stéphane Morucci, Sévan Azé, and Sébastien Salladin. A lean transformer model for dynamic malware analysis and detection, 2024.

[33] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H. Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, December 2015.

[34] Rami Sihwail, Khairuddin Omar, and Khairul Akram Zainol Ariffin. A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis. *International Journal on Advanced Science Engineering and Information Technology*, 8:1662, 09 2018.

[35] Jonathan Jiang and Mark Stamp. Multimodal techniques for malware classification, 2025.

[36] Sadia Nazim, Muhammad Alam, Safdar Rizvi, Jawahir Mustapha, Syed Hussain, and Mazliham Su'ud. Multimodal malware classification using proposed ensemble deep neural network framework. *Scientific Reports*, 15, 05 2025.

[37] Jane Bromley, James Bentz, Leon Bottou, Isabelle Guyon, Yann Lecun, Cliff Moore, Eduard Sackinger, and Rookpak Shah. Signature verification using a "siamese" time delay neural network. *International Journal of Pattern Recognition and Artificial Intelligence*, 7:25, 08 1993.

[38] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. *CoRR*, abs/1503.03832, 2015.

[39] Alexander Hermans, Lucas Beyer, and Bastian Leibe. In defense of the triplet loss for person re-identification. *CoRR*, abs/1703.07737, 2017.

[40] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering, 2010.

[41] Ian Shiel and Stephen O'Shaughnessy. Improving file-level fuzzy hashes for malware variant classification. *Digital Investigation*, 28:S88–S94, 2019.

[42] Ethan M. Rudd, David Krisiloff, Scott Coull, Daniel Olszewski, Edward Raff, and James Holt. Efficient malware analysis using metric embeddings. *Digital Threats*, 5(1), March 2024.

[43] Udbhav Prasad and Aniesh Chawla. A unified evaluation of learning-based similarity techniques for malware detection, 2026.

[44] Huijuan Wang, Boyan Cui, Quanbo Yuan, Ruonan Shi, and Mengying Huang. A review of deep learning based malware detection techniques. *Neurocomputing*, 598:128010, 2024.

[45] Microsoft. Pe format. `https://learn.microsoft.com/en-us/windows/win32/debug/pe-format`, 2025. Accessed: 2025-05-08.

[46] Xiang Ling, Lingfei Wu, Jiangyu Zhang, Zhenqing Qu, Wei Deng, Xiang Chen, Yaguan Qian, Chunming Wu, Shouling Ji, Tianyue Luo, Jingzheng Wu, and Yanjun Wu. Adversarial attacks against windows pe malware detection: A survey of the state-of-the-art. *Computers Security*, 128:103134, 2023.

[47] Mahmut KAYA and Hasan Şakir BİLGE. Deep metric learning: A survey. *Symmetry*, 11(9), 2019.

[48] Yikai Li, C. L. Philip Chen, and Tong Zhang. A survey on siamese network: Methodologies, applications, and opportunities. *IEEE Transactions on Artificial Intelligence*, 3(6):994–1014, 2022.

[49] Benjamin Midtvedt, Jesús Pineda, Henrik Klein Moberg, Harshith Bachimanchi, Joana B. Pereira, Carlo Manzo, and Giovanni Volpe. *Deep Learning Crash Course: A Hands-On, Project-Based Introduction to Artificial Intelligence*. No Starch Press, 2024.

[50] 3blue1brown. Attention in transformers, step-by-step | deep learning chapter 6, 2024.

[51] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[52] Wen Tai, H. T. Kung, Xin Dong, Marcus Comiter, and Chang-Fu Kuo. exBERT: Extending pre-trained models with domain-specific vocabulary under constrained training resources. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1433–1439, Online, November 2020. Association for Computational Linguistics.

[53] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[54] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019.

[55] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. ELECTRA: pre-training text encoders as discriminators rather than generators. *CoRR*, abs/2003.10555, 2020.

[56] Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, Nathan Cooper, Griffin Adams, Jeremy Howard, and Iacopo Poli. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference, 2024.

[57] Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, Nathan Cooper, Griffin Adams, Jeremy Howard, and Iacopo Poli. Finally, a replacement for bert, 2024. Hugging Face Blog.

[58] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *CoRR*, abs/1908.10084, 2019.

[59] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[60] David Davies and Don Bouldin. A cluster separation measure. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-1:224 – 227, 05 1979.

[61] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008.

[62] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[63] Michael Lester. Pe malware machine learning dataset. https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/, 2021. Practical Security Analytics LLC. Accessed: 2026-05-11.

[64] MITRE. MITRE ATT&CK. https://attack.mitre.org/, 2026. Accessed: 2026-05-29.

[65] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016.

[66] Suchin Gururangan, Ana Marasovic, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. Don't stop pretraining: Adapt language models to domains and tasks. *CoRR*, abs/2004.10964, 2020.

[67] Kexin Wang, Nils Reimers, and Iryna Gurevych. TSDAE: using transformer-based sequential denoising auto-encoder for unsupervised sentence embedding learning. *CoRR*, abs/2104.06979, 2021.

A

EMBER features

Listing A.1: Collapsed overview of one EMBER feature representation. Only representative fields and abbreviated samples are shown.


```
{
  "histogram":{
    "length":256,
    "sample":[54733.0,8020.0,5960.0,...,2996.0,4083.0]
  },

  "byteentropy":{
    "length":256,
    "sample":[22510.0,0.0,0.0,...,97545.0,109315.0]
  },

  "strings":{
    "numstrings":4029,
    "avlength":5.70,
    "entropy":6.55,
    "printables":22979,
    "printabledist":{
      "length":96,
      "sample":[395.0,240.0,218.0,...,228.0,200.0]
    }
  },

  "general":{
    "size":966594,
    "vsize":2072576,
    "imports":2,
    "exports":0,
    "has_debug":0,
    "has_relocations":0
  },

  "header":{
    "coff":{
      "timestamp":1504401044,
      "machine":"I386"
    },
    "optional":{
      "subsystem":"WINDOWS_GUI",
      "magic":"PE32",
      "major_image_version":40,
      "major_linker_version":6
```

```
    }
  },

  "section":{
    "entry":"...",
    "sections":[
      {"name":".rsrc","size":16384,"entropy":7.52},
      {"name":"nhgxrnnn","size":831488,"entropy":7.86},
      "..."
    ]
  },

  "imports":{
    "kernel32.dll":["lstrcpy"],
    "comctl32.dll":["InitCommonControls"]
  },

  "exports":[],

  "datadirectories":[
    {"name":"IMPORT_TABLE","size":149},
    {"name":"RESOURCE_TABLE","size":27980},
    "..."
  ]
}
```

B

Manually Extracted Dynamic Features

Table B.1: Overview of extracted network features, including their description and motivation. Many of these features are inspired by prior work, for example [5]. Note that one report corresponds to one malware sample.

Feature	**Description**	**Reasoning**
Packet counts from all flow events (received, transmitted, and aggregate)	Total number of packets observed across all flow events, including packets received from and transmitted to external sources, as well as the summed (aggregate) value of received and transmitted packets. Note that each component is used as a separate input feature.	Differences in packet counts can reflect distinct communication patterns across samples. A higher number of received packets may indicate extensive inbound communication, such as payload delivery, while a higher number of transmitted packets may indicate outbound communication, such as data exfiltration. The aggregate packet count provides a measure of overall communication, which can help distinguish between low-activity and high-activity behaviors across malware families.

Feature	Description	Reasoning
Byte size from all flow events (received, transmitted, and aggregate).	Total volume of data in bytes observed across all flow events, including received and transmitted byte sizes, as well as the summed (aggregate) value of received and transmitted bytes. Note that each component is used as a separate input feature.	Similar to the packet counts, differences in byte volumes can capture variations in data exchange patterns. Larger received byte volumes may indicate inbound transfers such as payload downloads, while larger transmitted byte volumes may indicate outbound transfers such as data exfiltration. The aggregate byte size summarizes total data transfer magnitude, which helps differentiate between samples with low network activity and those showing high activity.
Mean bytes per packet from all flow events (received, transmitted, and aggregate).	Mean bytes per packet across all flow events, computed separately for received, transmitted, and aggregate traffic, where the aggregate value is derived from the sum of received and transmitted bytes and packets within each flow. Calculated by finding the byte per packet for one flow event, summing all of them across all flow events, and dividing by the total amount of flow events. Note that each component is used as a separate input feature.	Anomalous mean bytes per packet values, across received, transmitted, and aggregate traffic, may capture unusual distinctive communication patterns or obfuscation attempts across malware families.

Feature	Description	Reasoning
Standard deviation of bytes per packet from all flow events (received, transmitted, and aggregate).	Standard deviation of bytes per packet across all flow events, computed separately for received, transmitted, and aggregate traffic, where the aggregate value is derived from the sum of received and transmitted bytes and packets within each flow. Note that each component is used as a separate input feature.	High variability in bytes per packet values, across received, transmitted, and aggregate traffic, may indicate irregular communication patterns or unusual network behavior; which can help distinguish between different samples or malware families.
Man inter-arrival time between flow events.	Mean time between consecutive flow events across all observed such events in the dynamic report.	Different inter-arrival time provides information about the nature of communication, where different patterns may indicate different types of malicious behavior.
Standard deviation of inter-arrival time between two flow events.	Standard deviation of the time between consecutive flow events across all observed such events in the dynamic report.	High variability can point to unusual communication patterns, which may also be indicative of different types of malicious behavior.
Amount of protocols from all flow events.	The total amount of UDPs and TCPs over all flow events in a report. Note that each amount is used as a separate input feature.	Difference in the transport protocol show different communication methods. Since TCP is used for more reliable communication, UDP prioritizes speed over reliability, the amount of each can be discriminative for different mawlare families.
Protocol ratios from all flow events .	The ratio of the total amount of UDPs and TCPs to the total number of packets across all flow events. The total number of packets is defined as the sum of TCP and UDP packets. Each ratio is used as a separate input feature.	The UDP and TCP ratios can be seen as the ratio of connectionless vs connection-oriented communication for all network activity, which can show unusual patterns as well as indicate different types of behavior.

Feature	Description	Reasoning
Amount of unique source IPs from all flow events.	The number of distinct source IP addresses observed across all flow events in the dynamic report.	The amount of unique source IP addresses can suggest unique malware behavior, like distributed or botnet attacks, indicating certain types of malware families.
Amount of unique destination IPs from all flow events	The number of distinct destination IP addresses observed across all flow events in the dynamic report.	The amount of unique destination IP addresses can also suggest unique malware behaviors, such as data exfiltration, pointing to certain types of malware families.
Amount of unique protocols used from all flow events	Checks how the amount of different protocols are used, essentially meaning if a sample uses UDP, TCP or both.	A variety of different protocols can indicate a different type of pattern than if only one of them is used, which may indicate a different type of malware behavior.
Amount of domains	The total amount of domains found in a report, where each is counted towards a total sum of all domains.	The total amount of domains shows different network activity patterns between samples, which can indicate different malware behaviors.
Amount of unique domains	The total amount of unique domains in a report, meaning that each new and unique domain is counted towards a total sum.	The amount of unique domains can indicate different levels of communication levels, which can also indicate different malware behaviors
Amount of different TLDs	The total amount of different TLDs found in a report (e.g ".com", ".net" etc). Each TLD found in a report is treated as an individual feature. The value of each TLD feature corresponds to the number of occurrences of that specific TLD in the report.	Different TLDs may indicate unique communication patterns and activity. For instance, frequent communication with country-specific TLDs could in some cases correlate with particular types of malware and malware behavior.

Feature	Description	Reasoning
Amount of DNS query types	The total amount of different DNS query types found in a report. Each query type found in a report is treated as an individual feature. The value of each query type feature corresponds to the number of occurrences of that specific query type in the report.	Different DNS query types reflect different forms of domain resolution and network lookup behavior. The existence and distribution of query types therefore provides additional information about how malware families attempt to resolve or interact with external resources.
Most common TCP port	The port that appears most frequently across all flow events where the transport protocol is TCP, determined by counting the occurrences of TCP ports in the report.	Common ports may indicate standard services, while unusual ports can indicate unusual behavior, indicating different types of malicious behavior.
Most common UDP port	The port that appears most frequently across all flow events where the transport protocol is UDP, determined by counting the occurrences of UDP ports in the report.	Exact same rationale as for TCP ports
Amount of unique TCP ports	The amount of unique TCP ports used in a report.	Indicates different levels and diversity of communication and network activity, which describes different types of malicious behavior.
Amount of unique UDP ports	The amount of unique UDP ports used in a report.	Same rationale as for TCP ports.
Maximum and minimum of TCP ports	The highest and lowest TCP port numbers observed across all flow events in the report. Note that each component is used as a separate input feature.	The range of ports used may indicate different network activity and communication attempts, which can describe different malicious behavior.

Feature	Description	Reasoning
Maximum and minimum of UDP ports	The highest and lowest UDP port numbers observed across all flow events in the report. Note that each component is used as a separate input feature.	Same rationale as for TCP ports.
Mean payload (byte) size across all flow events (received, transmitted, and aggregate)	Mean payload size is the mean bytes size across all flow events, computed separately for received, transmitted, and aggregate traffic, where the aggregate value is derived from the sum of received and transmitted bytes. Calculated by finding the byte for one flow event, summing all of them across all flow events, and dividing by the total amount of flow events. Note that each component is used as a separate input feature.	Similar rationale as for the mean bytes per packet feature.
Standard deviation of payload (byte) size	Standard deviation of bytes size across all flow events, computed separately for received, transmitted, and aggregate traffic, where the aggregate value is derived from the sum of received and transmitted bytes. Note that each component is used as a separate input feature.	Similar rationale as for the standard deviation of the bytes per packet feature.
Max and min of payload (byte) size	The smallest and biggest byte sizes found across all flow events in a report	Very large payloads may indicate substantial data transfers, while very small payloads may reflect different communication patterns. These patterns can help distinguish different types of network behavior across samples.

Feature	Description	Reasoning
Amount of HTTP methods	The different communication operations and methods used when transferring or collecting data over a network. This include "GET", "POST", "PUT", "DELETE", "HEAD", "PATCH", "OPTIONS". Each method found in a report is treated as an individual feature. The value of each method feature corresponds to the number of occurrences of that specific method in the report	Each method does a different operation when accessing a resource. The distribution of methods can therefore describe various communication patterns and help distinguish between different types of network and malicious behavior.

Table B.2: Signature features extracted from the dynamic reports.

Feature	Description	Reasoning
Memory and thread manipulation	Describes behaviours related to memory modification or thread control.	These behaviours may indicate malware that manipulates execution, for example through injection, unpacking, or process manipulation.
Process and object management	Describes behaviours related to creating, spawning, or interacting with processes and system objects.	Different malware families may create or interact with processes in different ways, which could make this behaviour useful for distinguishing different malicious patterns.
Registry access and modification	Describes behaviours related to opening, querying, creating, deleting, or modifying Windows registry keys and values.	Different registry use may indicate different types of malware behaviour.

Feature	**Description**	**Reasoning**
File access and modification	Describes behaviours related to opening, reading, writing, creating, or modifying files during execution.	File-system behaviour may describe how different samples interact with a device, since different malware types may read, create, or modify files differently.
Token and privilege management	Describes behaviours related to access tokens, privileges, or security-related permissions.	These behaviours may indicate attempts to access certain resources or change permissions, which could be more common in certain malware behaviours.
Network activity	Describes behaviours related to network communication, such as HTTP activity or connections to external IP addresses.	Network behaviour may indicate malware families that communicate with external infrastructure.
MITRE ATT&CK TTP signatures	Describes tactics, techniques, and procedures according to the MITRE ATT&CK framework.	TTP signatures provide higher-level behavioural categories that may vary between different malware families. Since a malware family can be associated with several TTPs, these signatures can provide discriminative information about the malicious behaviour. Note that each TTP signature found is used as a separate input feature.

Table B.3: Process features extracted from the dynamic reports.

Feature	**Description**	**Reasoning**
Process count	The total number of processes observed during sandbox execution.	Different malware families may spawn varying numbers of processes, which may be a discriminative indicator of malware behavior.
Process runtime (min, max, mean)	The minimum, maximum, and average duration of individual process lifetimes. Note that each component is used as a separate input feature	Runtime characteristics may reflect different behavioral patterns, which could differ across malware families.
Unique image executables	The number of distinct executable files observed across all processes.	Different malware samples may start different amounts of programs, which could help distinguish their behavior.
Parent-child process features	Describes the parent-child relationships between processes, including the average and maximum number of children per process, the child-to-parent ratio, and the number of unique parent processes. Note that each component is used as a separate input feature	Process spawning patterns may reveal how different malware launches different processes, where different families may have different spawning patterns
Process tree depth	Describes the length of the longest chain of processes spawned from an original process.	Different malware families may create short or deep process chains, describing how complex their execution behavior is and may be discriminative for certain malicious behaviors.

Table B.4: Dumped file features extracted from the dynamic reports.

Feature	**Description**	**Reasoning**
Dumped file count	The total number of files or objects dumped during sandbox execution.	Different malware samples may dump, unpack, or create different numbers of files, which indicate certain behaviors.
Dumped file size (min, max, mean)	The minimum, maximum, and mean size of the dumped files. Note that each component is used as a separate input feature	Different malware families may create different sizes of dumped files may differ, which may capture distinctive behavior across these families.

C

Dynamic Report Sections

Listing C.1: Simplified excerpt from the network section of a dynamic report. The section contains several subsections, including `flows`, `requests`, and `ips`. Each subsection contains multiple entries, but only one representative entry is shown.

```
'flows': [
    {
        'id': 2,
        'src': '10.127.0.40:61844',
        'dst': '8.8.8.8:53',
        'proto': 'tcp',
        'pid': 1912.0,
        'procid': 34.0,
        'first_seen': 1601.0,
        'last_seen': 1985.0,
        'rx_bytes': 398,
        'rx_packets': 5,
        'tx_bytes': 327,
        'tx_packets': 6,
        'protocols': array(['dns'], dtype=object),
        'domain': 'settings-win.data.microsoft.com',
        'tls_ja3': None,
        'tls_ja3s': None,
        'tls_sni': None,
        'country': None,
        'as_num': None,
        'as_org': None
    },
    ...
],

'requests': [
    {
        'flow': 2,
        'index': 1,
        'at': None,
        'dns_request': {
            'domains': array(['settings-win.data.microsoft.com'], dtype=object),
            'questions': array([
                {
                    'name': 'settings-win.data.microsoft.com',
                    'type': 'IN A',
                    'value': None
                }
            ], dtype=object)
        },
        'dns_response': None,
        'http_request': None,
        'http_response': None
    },
    ...
],

'ips': [
    ('13.71.55.58', {'asn': 'AS8075', 'cc': 'IN'}),
    ...
]
```

Listing C.2: Simplified excerpt from the process section of a dynamic report. Only one representative process entry is shown.

```
'processes': [
    {
        'procid': 83,
        'procid_parent': 56.0,
        'pid': 736,
        'ppid': 3432,
        'cmd': '"C:\\Users\\Admin\\AppData\\Local\\Temp\\f922e99455edbe97471c69e9b01a3a35a51753bae40235ef9ce141f536678be1.exe"',
        'image': 'C:\\Users\\Admin\\AppData\\Local\\Temp\\f922e99455edbe97471c69e9b01a3a35a51753bae40235ef9ce141f536678be1.exe',
        'orig': False,
```

```
        'started': 125,
        'terminated': 1141.0
    },
    ...
]
```


Listing C.3: Simplified excerpt from the analysis section of a dynamic report. Although the section contains several metadata fields, only the `tags` and `ttp` fields are used in this work.


```
'analysis': {
    'backend': 'sbx7rf183',
    'features': array(['analog', 'overview'], dtype=object),
    'platform': 'windows10-2004_x64',
    'reported': '2025-04-29T20:34:12Z',
    'submitted': '2025-04-29T08:45:02Z',
    'score': 10,
    'tags': array(['family:neconyd', 'discovery', 'trojan'], dtype=object),
    'ttp': array(['T1614.001'], dtype=object),
    ...
}
```


Listing C.4: Simplified excerpt from the signature section of a dynamic report. Only one representative signature entry is shown.


```
'signatures': [
    {
        'label': 'system_language_discovery',
        'name': 'System Location Discovery: System Language Discovery',
        'score': 3.0,
        'ttp': array(['T1614.001'], dtype=object),
        'tags': array(['discovery'], dtype=object),
        'indicators': array([
            {
                'ioc': '\\REGISTRY\\MACHINE\\SYSTEM\\ControlSet001\\Control\\NLS\\Language',
                'description': 'Key opened',
                'at': None,
                'pid': None,
                'procid': 86.0,
                'pid_target': None,
                'procid_target': None,
                'flow': None,
                'stream': None,
                'dump_file': None,
                'resource': None,
                'yara_rule': None
            },
            {
                'ioc': '\\REGISTRY\\MACHINE\\SYSTEM\\ControlSet001\\Control\\NLS\\Language',
                'description': 'Key opened',
                'at': None,
                'pid': None,
                'procid': 88.0,
                'pid_target': None,
                'procid_target': None,
                'flow': None,
                'stream': None,
                'dump_file': None,
                'resource': None,
                'yara_rule': None
            },
            ...
        ], dtype=object)
    },
    ...
]
```


Listing C.5: Simplified excerpt from the memory dumped section of a dynamic report. Only representative entries are shown.


```
'dumped': [
    {
        'at': 157,
        'pid': 736,
        'procid': 83.0,
        'path': None,
        'name': 'memory/736-0-0x0000000000400000-0x0000000000424000-memory.dmp',
        'kind': 'region',
        'origin': 'exception',
        'addr': 4194304.0,
        'length': 147456.0,
        'md5': None,
        'sha1': None,
        'sha256': None,
```

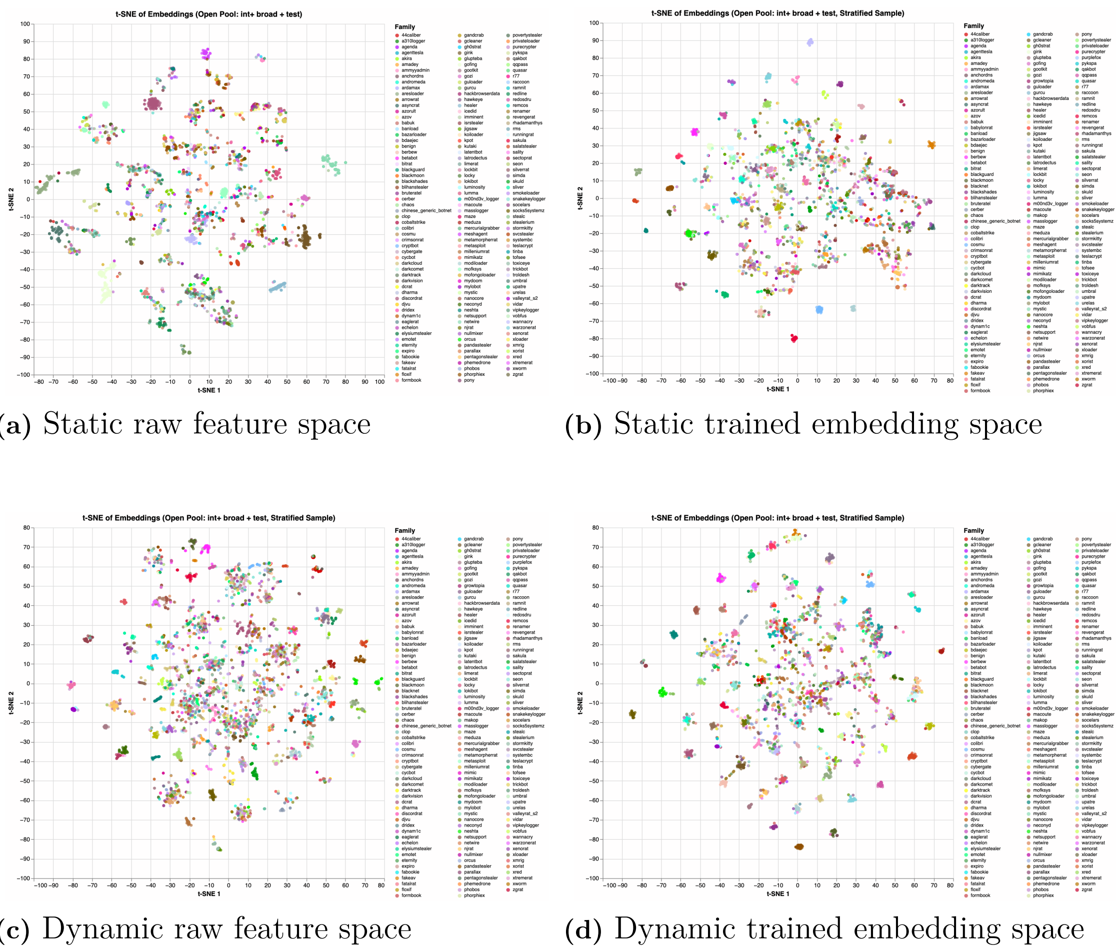
```
        'sha512': None,
        'ssdeep': None,
        'size': None
    },
    {
        'at': 469,
        'pid': 1864,
        'procid': 86.0,
        'path': 'C:\\Users\\Admin\\AppData\\Roaming\\omsecor.exe',
        'name': 'files/0x0008000000023d24-10.dat',
        'kind': 'martian',
        'origin': 'martian',
        'addr': None,
        'length': None,
        'md5': '50d23118ecc99178fd7bba512dfdb975',
        'sha1': '687f52fcf52f445501e2c3612dcb427d814bc42a',
        'sha256': '3c7c36401a5c6b985215b0008dfcad87bdba0d5e5c29414daf49f2a0f24730f1',
        'sha512': '5bf6ed33de7b740ec5d6997facfa3bfb1a9bcf86b69-
        70d98338cd5ba2f4c509000b2bdcb0a9be-
        778a0f8c8b9f5ad17132e7d180f21c73a68bcd6d67ec03b7090',
        'ssdeep': '1536:jDfDbhERTatPLTHOiqNZg3mqKv6yORrwFd1tSEsF27da6ZW72Foj/MqMabadwCi3:viRTeHOiqAW6J6f1tqF6dngNmaZCiaI',
        'size': 137285.0
    },
    ...
]
```


Listing C.6: Simplified excerpt from the extracted section of a dynamic report. Only one representative entry is shown.


```
'extracted': [
    {
        'dumped_file': 'memory/800-1-0x0000000000400000-0x0000000000429000-memory.dmp',
        'resource': 'behavioral1/memory/800-1-0x0000000000400000-0x0000000000429000-memory.dmp',
        'config': {
            'family': 'neconyd',
            'tags': None,
            'rule': 'Neconyd',
            'c2': array(None, dtype=object),
            'version': None,
            'botnet': None,
            'campaign': None,
            'mutex': None,
            'decoy': None,
            'wallet': None,
            'dns': None,
            'keys': None,
            'webinject': None,
            'command_lines': None,
            'listen_addr': None,
            'listen_port': None,
            'listen_for': None,
            'shellcode': None,
            'extracted_pe': None,
            'credentials': None,
            ...
        },
        'path': None,
        'ransom_note': None,
        'dropper': None,
        'credentials': None
    },
    ...
]
```

D
Additional Results

In Figure D.1 one can see the single modality tSNE plots before and after Siamese learning.



(a) Static raw feature space

(b) Static trained embedding space

(c) Dynamic raw feature space

(d) Dynamic trained embedding space

Figure D.1: t-SNE visualizations of the open-set evaluation space for static and dynamic representations before and after Siamese training. Notice the well separated clusters in the trained files, those are the unseen families from families seen during training. The unseen families struggle to form meaningful separated clusters.

DEPARTMENT OF PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY