Demystifying Feature Engineering in Malware Analysis of API Call Sequences

Tianheng Qu[1,2], Hongsong Zhu[1,2], Limin Sun[1,2], Haining Wang[3],
Haiqiang Fei[1], Zheng He[4], Zhi Li[1,2]
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3]The Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, USA
[4]National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, China
{qutianheng, zhuhongsong, sunlimin, feihaiqiang, lizhi}@iie.ac.cn, hnw@vt.edu, hezheng@cert.org.cn

***Abstract*—Machine learning (ML) has been widely used to analyze API call sequences in malware analysis, which typically requires the expertise of domain specialists to extract relevant features from raw data. The extracted features play a critical role in malware analysis. Traditional feature extraction is based on human domain knowledge, while there is a trend of using natural language processing (NLP) for automatic feature extraction. This raises a question: how do we effectively select features for malware analysis based on API call sequences? To answer it, this paper presents a comprehensive study of investigating the impact of feature engineering upon malware classification. We first conducted a comparative performance evaluation under three models, Convolutional Neural Network (CNN), Long Short-Term Memory (LSTM), and Transformer, with respect to knowledge-based and NLP-based feature engineering methods. We observed that models with knowledge-based feature engineering inputs generally outperform those using NLP-based across all metrics, especially under smaller sample sizes. Then we analyzed a complete set of data features from API call sequences, our analysis reveals that models often focus on features such as handles and virtual addresses, which vary across executions and are difficult for human analysts to interpret.**

***Index Terms*—malware, feature engineering, machine learning, deep learning**

I. Introduction

The number of malware attacks has steadily increased over the past decade. In 2024, AV-TEST Institute has recorded over 450,000 new malware variants daily [1], making it challenging to rely solely on researchers' manual analysis to manage this vast volume. Leveraging ML to analyze malware API call sequences has been widely regarded as an effective approach with promising results in practice [2]–[5]. This approach offers an easily deployable and highly scalable solution for large-scale malware analysis. In the application of ML analysis, feature extraction is critical to model performance and we need to identify features suitable for classification [2], [6]. The effective feature extraction enables the model to capture genuine and unique characteristics and behaviors, thereby distinguishing between benign and malicious activities more accurately. This process often relies on the expertise of domain specialists [7]–[9], requiring them to manually select potentially valuable features and transform raw data into structured feature vectors.

On one hand, the efficiency of feature extraction is always desirable. Due to the time-consuming and complex nature of manual feature extraction (i.e., knowledge-based), some research treats APIs as sequences of words and incorporates NLP techniques to handle them, achieving promising results [10], [11]. Compared to those methods that rely on knowledge-based feature engineering, such an NLP-based approach is simpler. This is because neural networks are enabled to automatically learn and capture behavioral features or heuristic rules, and feature can be directly input into models. Nevertheless, whether we can fully rely on NLP-based approaches for accurate classification without manual feature extraction remains an open question.

On the other hand, the completeness of feature extraction is critical for accurate classification, but it is challenging to achieve high accuracy in the real world [2]. Due to the heterogeneous types of API parameters, lack of contextual patterns, and semantic complexity [12], manual feature extraction on API call sequences often prioritizes features deemed important based on domain knowledge [2], [13], [14]. However, features that are less recognizable by human domain experts may be overlooked [2]. For example, we tend to focus on string features while frequently neglecting virtual address features, as virtual addresses are more difficult for humans to interpret. This discrepancy may introduce potential biases in feature engineering.

Moreover, when designing ML models, we typically set specific scenario objectives, aiming for the learned correlations to align with those objectives. However, artifacts unrelated to the security problem create shortcut patterns for separating classes, causing the model to rely on these artifacts rather than focusing on the actual task [2], [27]. For example, in a network intrusion detection system, where a large fraction of attacks in the dataset originate from a certain network region. The model may learn to detect a specific IP range, instead of generic attack patterns [27]. Compounded by the black-box nature of ML models, spurious correlations often remain unidentified,

TABLE I: Related work on malware analysis based on API call sequences.

Authors	Feature engineering	Parameter types	Embedding method	Model
Kolosnjaji. [15]	Knowledge-based	APIs	One-hot	CNN + LSTM
Zhang et al. [16]	Knowledge-based	APIs	One-hot	LSTM
Zhang et al. [12]	Knowledge-based	String and Integer	Hash trick	CNN + LSTM
Agrawal et al. [17]	Knowledge-based	String	N-gram	N-gram + LSTM
Salehi et al. [18]	Knowledge-based	String and return value	Short text clustering	SVM
Li et al. [19]	Knowledge-based	String and Integer	hash + similarity encoder	GINE + GAT
Chen et al. [20]	Knowledge-based	String	One-hot	TextCNN / Transformer
Chen et al. [21]	Knowledge-based	String	Short text clustering	TextCNN / BiLSTM
Liu et al. [22]	Knowledge-based	String	Short text clustering	Meta-graph + GAT
Rosenberg et al. [23]	Knowledge-based	String	One-hot	RNN / CNN
Tian et al. [24]	NLP-based	String	One-hot	Random Forest
Jindal et al. [11]	NLP-based	String	One-hot	CNN + BiLSTM + Attention
Karbab et al. [25]	NLP-based	String	N-gram	XGBoost
Liu et al. [26]	NLP-based	String and Integer	Short text clustering	GNN
Trizna et al. [10]	NLP-based	String and Integer	Subword embedding	Transformer
Proposed Method	Both	**All and virtual address**	**Hash trick**	**CNN / LSTM / Transformer**

which cause misjudgments on the model's applicability and limitations, as well as overly optimistic evaluations. These problems, and the importance of feature extraction for model analysis, motivate us to conduct a comprehensive study on the feature engineering in malware analysis, aiming to address the following questions:

RQ1. Under the same conditions, is there a performance difference between knowledge-based and NLP-based feature engineering?

RQ2. Which features significantly impact classification accuracy in malware analysis based on API call sequences?

RQ3. Due to the presence of unrelated artifacts, is the model truly focused on the actual task as we intended? What features does it actually use to differentiate malware?

By applying knowledge-based and NLP-based feature engineering methods to the open-source AV-TEST dataset [28], we processed the complete set of API call sequence features and fed them into three ML models (CNN, LSTM, and Transformer) to compare the performance of the two approaches across different models. Then we analyzed the impact of various feature combinations on classification accuracy. Finally, we employed interpretability methods to identify spurious correlations and unveiled the features that the models truly rely on.

Our key findings include (1) knowledge-based feature engineering methods consistently outperform NLP-based ones, particularly when dealing with small sample datasets; (2) regardless of the feature engineering method being applied, CNN models consistently demonstrate the best performance; (3) there is a stark contrast in the performance of knowledge-based and NLP-based methods when increasing the number of input features; (4) our analysis reveals that models often focus on features such as handles and virtual addresses, which vary across executions and are difficult for human analysts to interpret. Thus, effective feature engineering methods such as feature hashing [29] are needed to improve the generalizability of these features.

The rest of this paper is structured as follows. Section II introduces related work on malware analysis using API call sequences. Section III details the methodology of our study. Section IV describes the experimental setup. Section V presents the experimental results and analysis. Section VII discusses how we address the threats to validity, along with recommendations and limitations of this study, and finally, SectionVIII concludes.

II. RELATED WORK

This section surveys prior research on using API call sequences for malware analysis. Table I summarizes the design and processing strategies used in these studies, where feature engineering indicates what type it was applied, parameter types denote the types of features involved, embedding method specifies the feature embedding approach, and model refers to the neural network model employed. In the following, we first describe knowledge-based feature engineering, and then we present NLP-based feature engineering approaches.

When employing knowledge-based feature engineering methods, API sequence features are often among the core features. Kolosnjaji et al. [15] used API sequences as input and applied a filter of size three to capture patterns across three consecutive APIs. They processed these features using a CNN, and utilized an LSTM to handle temporal sequence characteristics. Similarly, Zheng et al. [16] constructed behavioral chains based on API call sequences and used an LSTM to detect malicious behaviors.

Some previous studies incorporate API parameter features into the model. Zhang et al [12] used a hashing algorithm to map various types of API parameters (such as paths, registry

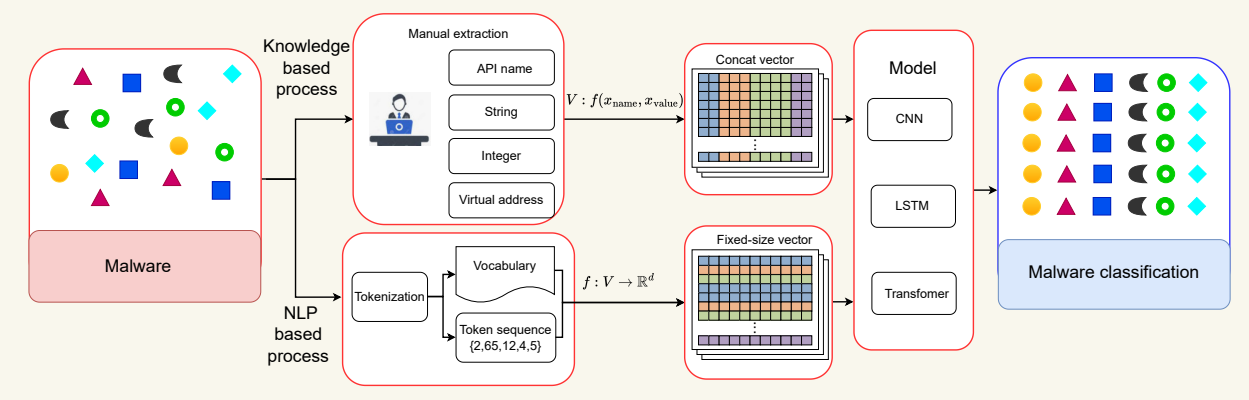


Fig. 1: Overview of Feature Extraction and Malware Classification.

keys, URLs, and IPs) along with API names and categories into feature vectors, which were then used to train a deep neural network for malware detection. Agrawal et al. [17] mapped the top K high-frequency N-grams of API names and parameter strings into feature vectors and applied a stacked LSTM model to detect malware.

Salehi et al. [18] specifically processed return values, representing calls as (API, variable, return value) tuples, and identified a set of selective and distinctive features, which were classified using a Support Vector Machine (SVM). Li et al. [19] proposed a hybrid feature encoder to extract semantic features from API names and parameters, and then they constructed an API call graph to incorporate structural features.

Auxiliary feature recognition or heuristic rules have also been introduced to enhance the capacity of feature representations. For example, Chen et al. [20] extracted Indicators of Compromise (IOCs) from Cyber Threat Intelligence (CTIs) and used IOCs to assist in identifying the security sensitivity level of API calls. Chen et al. [21] assessed the sensitivity of runtime parameters by using heuristic rules, and integrated this sensitivity into the learning process of API call sequences. They constructed two independent networks for feature learning, respectively, based on 2D convolution and LSTM.

Moreover, some studies have adopted adversarial approaches to enhance model robustness. Liu et al. [22] addressed the issue of missing structural information and improved adversarial robustness by constructing a sensitivity-based heterogeneous graph to characterize malware execution behaviors. Rosenberg et al. [23] proposed a black-box adversarial attack method that generates perturbed API calls to evade malware detection while maintaining the integrity of the malware's functionality.

In NLP applications, various tokenization techniques are used for modeling. For example, Tian et al. [24] treated API calls and their parameters as independent strings, using their frequency as features to train the model. Jindal et al. [11] applied document processing techniques to tokenize sandbox reports, extracting vocabulary and mapping a one-hot vector to each word, and then they used a combination of one-dimensional convolution, LSTM, and attention mechanisms to learn these representations.

Karbab et al. [25] proposed a Bag-of-Words (BoW) model to construct behavioral reports, automatically extracting relevant security features to facilitate malware detection and attribution without investigators' intervention. Similarly, Liu et al. [26] incorporated heuristic rules to evaluate the sensitivity of each API event parameter, sampling suspicious API events to improve feature representation. In addition, Trizna et al. [10] used integrated gradients and attention activation techniques to analyze the performance of self-supervised Transformer models in malware detection and classification.

Our research focuses on extracting and analyzing all features within API call sequences under both knowledge-based and NLP-based feature engineering conditions, evaluating their performance on different models to better understand the effectiveness of extracted features for malware analysis.

III. Methodology

As shown in Fig. 1, there are two approaches for processing raw API call sequences into feature vectors suitable for model input. One approach is to use manual feature extraction to convert the data into a structured format. The feature extraction relies on expert knowledge to manually identify key features from datasets, followed by the application of vectorization techniques to transform these features into vector representations. Such a knowledge-based approach is detailed in Section III-B. The other is the NLP-based approach. It borrows concepts from the field of document classification, treating them as a sequence of words, through tokenization, a vocabulary and token sequence is obtained, and then an embedding layer is used to convert the samples into vector representations. This method leverages neural networks to replace the hand-crafted heuristics, thereby learning these behavioral patterns or heuristic rules. This NLP-based approach is detailed in Section III-C.

Notably, these two approaches adopt distinct strategies for organizing feature vectors. As shown in Fig. 1, the knowledge-

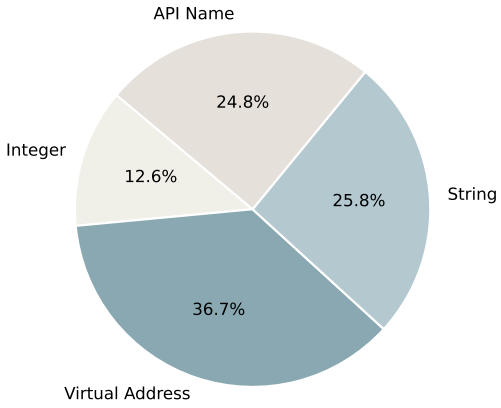


Fig. 2: Proportional distribution of different data types across the API sequences of all samples in the dataset.

based approach encodes different types of features separately and concatenates them by field to form the final input vector. For example, dimensions 0–31 of the input vector correspond to API names, while dimensions 32–96 represent string parameters. Such an organization explicitly preserves type boundaries and structural information inherent in the raw data. In contrast, the NLP-based approach treats both API names and their parameters as a unified token sequence, requiring the model to implicitly learn the distinctions among data types from contextual cues.

Once the API call sequence has been converted into vector representations, various models can be employed for training to perform specific tasks (such as malware classification). A detailed description on the application of specific models is given in Section III-D.

A. Data Characterization

The API call sequence reflects the interactions between a sample and its environment, enabling the extraction of rich information. The observed data points need to be represented in a format suitable for processing by ML models [30]. Existing representations focus on extracting API names from the sequence and using them as function call features for model input [31]–[33], which have yielded promising detection results. Some studies have further explored extracting API parameter features. However, due to the diverse types of API parameters, the lack of contextual patterns, and the complexity of their semantics, extracting the semantic features of API parameters remains a significant challenge. Currently, related research still focuses on feature extraction for string and integer parameters.

However, if the data representation does not contain the information required to answer the question (e.g., whether a specific memory operation is benign or malicious), it will result in an irreducible error. As shown in Fig. 2, a statistical analysis of different data types in the dataset reveals that the virtual address has the highest proportion among all types, nearly equivalent to the combined proportions of API name and integer. It implies a large amount of virtual address data in the API call sequences, and discarding this feature may result in the loss of important semantic information. Therefore, we retained the virtual address in feature construction. Ultimately, four feature types are extracted from the API call sequences: (1) API name, (2) string parameter, (3) integer parameter, and (4) virtual address parameter. These features encompass all data types within the API call sequences, allowing for further investigation in the ablation study in Section V-A to explore the impact of different feature combinations on model generalization performance. In the following, we present API call sequence examples in the dataset and their corresponding feature types.

1) API Name: API names are usually composed of multiple words, with the first letter of each word capitalized. For example, the API name `LdrGetProcedureAddress` is shown in Fig. 3. Windows API naming typically follows strict conventions and specific patterns [34]. Extracting API names as features effectively captures semantic information in function calls, which is crucial for context-based analysis and pattern recognition.

2) String Parameter: Strings in API parameters typically reference resources such as filenames, registry keys, dynamic libraries, domain names, and shell commands. As shown in Fig. 3, the first parameter `ModuleName` is `IMM32.DLL` indicating a reference to the `IMM32.DLL` dynamic library.

3) Integer Parameter: The semantic meaning of integer values needs to be interpreted in conjunction with the parameter name, as the same integer value may convey entirely different meanings under a different parameter name. As shown in Fig. 3, the fourth parameter `Ordinal` with a value set to zero, indicating that the function is not called by ordinal number. However, when the parameter name is `Size`, the value zero indicates the buffer size.

4) Virtual Address Parameter: In Windows API, virtual addresses are associated with handle types in the system, representing addresses that point to specific resources or objects. As shown in Fig. 3, the second parameter `ModuleHandle` represents the base address of the `IMM32.DLL` module in memory, while the last parameter `FunctionAddress` indicates the memory address of the loaded `ImmCreateContext` method. Due to the dynamic nature and high-dimensional sparsity of virtual address values, there is almost no research that incorporates virtual address type parameters as features in models.

B. Knowledge-based Feature Engineering

Most ML-based malware detectors rely heavily on domain-specific expertise [7], [8], [11], which requires malware experts to manually select valuable features and transform raw data into structured feature vectors. Although techniques such as Recursive Feature Elimination (RFE) [35], Principal Component Analysis (PCA) [36], and tree-based feature selection methods (e.g., Random Forest, XGBoost) can be applied, some

```
{
 "api": "LdrGetProcedureAddress",
 "arguments": [
  {
   "name": ModuleName,
   "value":ADVAPI32.dll,
  },
  {
   "name": ModuleHandle,
   "value":0x76520000,
  },
  {
   "name": FunctionName,
   "value":CryptDeriveKey,
  },
  {
   "name": Ordinal,
   "value":0,
  },
  {
   "name": FunctionAddress,
   "value":0x76563464,
  }]
}
```


Fig. 3: An example of an API call sequence report.

potentially effective features may be overlooked due to the limitations of domain knowledge. Furthermore, RFE and PCA primarily focus on the relationships between features or select features based on variance or principal components, without directly considering the correlation between features and the target variable. Consequently, these methods may miss some features that are crucial for the classification task.

To avoid missing any potentially valuable features that are listed in Table II, we perform feature encoding on all the data in the API calls and concatenate the encoded feature vectors, which are then provided as fed to the model. Then, we apply Explainable Artificial Intelligence (XAI) methods to directly calculate the relationships between the features and the target variable. Based on the API data characterization described in Section III-A, appropriate processing methods are applied to different data types. In the following, we detail the processing method for each data type.

1) API Name: Due to Word2Vec's effectiveness in capturing semantic relationships and contextual dependencies, it has been widely applied in malware detection for embedding API names [19], [37]. The API call sequences from all samples in the dataset are extracted and used to train a Word2Vec skip-gram model, resulting in a fixed-size vector space. After training, each distinct API name in the corpus is represented as a vector that captures its semantic relationships.

2) String Parameter: The diversity of strings makes it challenging to process them. To represent semantically similar strings as closely located vectors, we employed a similarity encoding method to extract feature vectors for these strings. First, the strings are converted into n-gram feature vectors, and a cosine similarity function is then used to measure the similarity between two strings for effectively capturing the similarity between strings, especially in the case of high-dimensional sparse vectors. The cosine similarity function is defined as follows:

TABLE II: Feature representation overview

Feature Sources	Feature Type		Encoding Method	Vector Size
API name	String		Word2Vec	32-dim
API Parameters	String	File Paths	Similarity encoding	16-dim
		DLLs name		16-dim
		Registry keys		16-dim
		URLs		16-dim
	Integer		Hashing trick	16-dim
	Virtual address			20-dim

$$\text{sim}(s_i, s_j) = \frac{G(s_i) \cdot G(s_j)}{\|G(s_i)\| \times \|G(s_j)\|} \tag{1}$$

where $G(s_i)$ and $G(s_j)$ are the n-gram feature vector representations of strings s_i and s_j. These vectors are constructed by extracting 3-grams, 4-grams, and 5-grams to form a consecutive character n-gram set for each string variable, i.e., $G(s) = G_3(s) \cup G_4(s) \cup G_5(s)$. $G(s_i) \cdot G(s_j)$ denotes the dot product of the two vectors. $\|G(s_i)\|$ and $\|G(s_j)\|$ are the norms of the two vectors, respectively.

Given a training corpus C containing N strings ($C = \{s_1, s_2, \ldots, s_N\}$), the top K most frequent strings are extracted from the training corpus C to form a known set $D = \{d_1, d_2, \ldots, d_K\}$, where $D \subseteq C$. These strings are stored for use in the feature encoding phase. In the feature encoding phase, the strings in API parameters s_i are represented as a feature vector $[\text{sim}(s_i, d_1), \text{sim}(s_i, d_2), \ldots, \text{sim}(s_i, d_K)] \in \mathbb{R}^K$. Specifically, for each string s_i, we calculate its cosine similarity with each string in the known set D, resulting in a K-dimensional feature vector that represents the similarity of the string to the known set of strings.

According to previous studies [38]–[40], strings such as file paths, DLLs, registry keys, and URLs are considered to be the most critical ones. Therefore, we trained similarity encoders separately for these four types of strings, resulting in four independent encoders. During the feature encoding phase, we identified the type of each string through regular expression matching and applied the corresponding encoder for processing.

3) Integer and Virtual Address Parameters: Single integer values and virtual address values do not provide meaningful semantic information on their own; they need to be interpreted in the context of their parameter names. For example, the number 21 has different meanings when it is interpreted as a port versus as a size. To address this, we adopted the feature hashing method [29], which allows parameters with the same name to be mapped to the same bucket index, thereby ensuring their semantic similarity in the feature space, specifically, as shown in Formula 2:

$$\phi_{h,\xi_i}(X) = \sum_{j:h(x_{\text{name}j})=i} \xi(x_{\text{name}j}) \log(|x_{\text{value}_j}| + 1) \tag{2}$$

where x_{name_j} represents the parameter name and its possible additional information such as a memory segment, while x_{value_j} denotes the value corresponding to this parameter name.

Since the parameter values may be sparsely distributed over a range, we use a logarithmic function to normalize the values, which squashes the range.

For integer parameters, let X represent the list of API integer parameters, where each parameter $x_j \in X$ consists of a parameter name x_{name_j} and an integer value x_{value_j}. In this case, the feature hashing formula can be simplified to hash based only on the parameter name: $h(x_{\text{name}_j})$.

High-level additional information is used to improve the performance of an ML model [41]. To enhance the accuracy of feature representation for virtual address parameters, the feature hashing encoding considers both the parameter names and the memory segments. This is because different address segments may correspond to different functional modules or operations. We use a combination of the parameter name and memory segment information to determine the hash classification: $h(x_{\text{name}_j}, x_{\text{seg}_j})$. Here, x_{name_j} represents the parameter name, and x_{seg_j} denotes a specific memory segment, such as user space or kernel space.

C. NLP-based Feature Engineering

When applying NLP techniques to process API call sequences, we followed the standard model for document classification and performed the following steps.

1) Data Cleaning: The sample reports in the dataset are provided in JSON format. We removed special characters, including brackets and file path symbols (e.g., `[ ]`, `{ }`, `:`, `'`, `/`, etc.) from the JSON structure, ensuring that the data was transformed into a sequence of words.

2) Tokenization: Tokenization breaks down the input text into basic units, called tokens, which represent the input data in a form that a ML model can process. Common tokenization methods include Whitespace, Wordpunct [42], and Byte Pair Encoding (BPE) [43]. The Whitespace tokenization separates words based on spaces, tabs, and newline characters, while WordPunct further uses punctuation marks as delimiters. The core idea of BPE is to generate new subword units by recursively merging the most frequent pairs of characters. Initially, each character in the text is treated as a token; the algorithm then repeatedly merges the most frequent consecutive token pairs to form new tokens until a specified number of merges is reached or no further merges can be performed.

In the experiment, we evaluated three different tokenization methods, limiting the vocabulary size to the most frequent tokens within $Vocab \in \{30K, 50K, 70K\}$, and introduced two special tokens to represent all other tokens (`<unk>`) and padding for shorter sequences (`<pad>`). The experimental results are listed in Table III, showing that the F1 scores(F1) for the three methods are comparable. Due to the superior readability and intuitiveness of the Whitespace tokenization method, which facilitates subsequent analysis, we selected the Whitespace method and limited the vocabulary size to 70K.

3) Word Embedding: After tokenization, the text data is transformed into a sequence of tokens, from which a vocabulary containing all possible tokens is generated. Subsequently, the integer-represented token sequence (i.e., the index sequence) is passed through an embedding layer, converting it into embedding representations with predefined dimensions. These vectors can then be fed into subsequent models for training. Word embeddings capture semantic information, enabling the model to learn and leverage the contextual relationships between tokens.

TABLE III: Results of different tokenizer methods and vocabulary sizes.

Tokenizer	Vocab	ACC	PR	RC	F1
WordPunct	30K	89.19	**88.79**	**88.72**	**88.28**
	50K	**89.65**	**88.79**	88.70	88.24
	70K	93.08	92.73	90.47	90.93
Whitespace	30K	87.53	87.33	87.49	87.41
	50K	**93.99**	**89.64**	88.68	88.53
	70K	94.80	90.82	**89.08**	**89.69**
BPE	30K	90.40	90.30	90.49	90.41
	50K	**92.03**	**91.64**	91.68	91.53
	70K	94.44	93.57	**91.36**	**91.04**

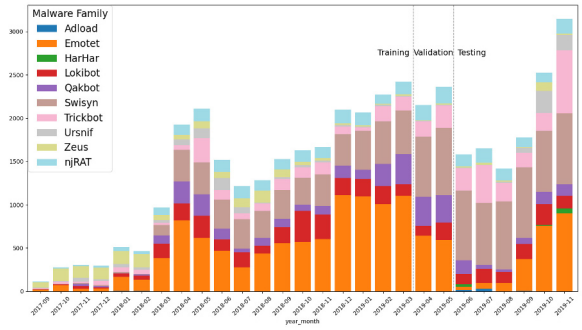
D. Choice of Models

ML is a set of statistical methods for automating data analysis and enabling systems to perform tasks on the data without being explicitly programmed for them. In the malware domain, typical tasks include binary classification [44], [45] and multiclass classification [46]. Our work focuses on classification tasks.

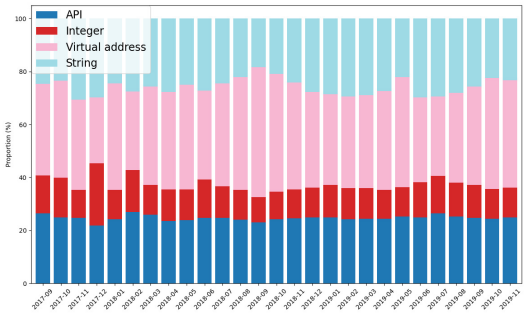
We selected three popular ML models: CNN, LSTM, and Transformers, because they are consistently among the best-performing classifiers evaluated in previous works (summarized in Table I). We evaluated their performance under different conditions, using either knowledge-based or NLP-based feature engineering. We also analyzed the specific contributions of various features to the classification task.

The goal of using CNN is to enable the model to gradually learn high-level feature representations within sequences, particularly focusing on the local feature patterns of sample behaviors. LSTM, which introduces gating mechanisms, effectively regulates the flow of information, thereby capturing and retaining long-term dependencies in sequences more accurately and better recognizing contextual interactions between API calls. Transformers significantly enhance the ability to model global dependencies within sequences through the self-attention mechanism. The self-attention mechanism allows the representation of each token to include not only its own information but also the information from other relevant tokens, effectively capturing global dependencies in the sequence. This is crucial for analyzing complex malicious behavior patterns that involve multiple consecutive API calls.

The output of the models described above, after being flattened, is passed into a multilayer perceptron (MLP). The MLP consists of fully connected layers. Following fully connected layers, a ReLU activation function and a Dropout mechanism are applied to reduce the risk of overfitting. We have listed the optimal hyperparameter settings for different models in the Appendix for reproducibility.



(a) The distribution of different malware families across each month, with data before May 2019 used for training and data after for testing.



(b) The monthly proportion changes of different features, representing the cumulative proportion of each feature type across all samples for that month.

Fig. 4: Re-splitting the dataset.

IV. Experimental Setup

A. Datasets

While there are several publicly available malware datasets, such as [47]–[52], these datasets do not fully align with the requirements of our study. For example, the datasets [47], [50] only contain static features, the datasets [49], [51], [52] provide only API call sequences without API parameters, and the malware samples in the dataset [48] are not executable.

In our experiments, we used the open-source Avast-CTU dataset [28], which comprises sandbox reports generated by CAPEv2 [53]. The report is in JSON format and includes sample hash values, API calls, network activity, file system operations, malicious behavior flags, etc. It contains approximately 44,000 malware, covering six types of malware from ten different malware families, with a collection period spanning from July 2021 to September 2021. Due to the absence of goodware in this dataset, we obtained Portable Executable (PE) files of goodware from mainstream download sites [54]–[56] and scanned them using VirusTotal [57] to ensure that the samples were non-malicious, thereby enhancing the accuracy of our evaluation. Moreover, due to the limited number of keylogger samples, we supplemented the dataset with additional keylogger malware from the VirusShare [58]. All collected files were executed in the same CAPEv2 sandbox environment. According to Kuchler [59], two minutes of execution are sufficient for malware to expose its malicious behaviors. Therefore, we set the execution time of the samples in the sandbox to five minutes to allow each sample sufficient time to expose its malicious behaviors as much as possible, resulting in a final dataset comprising 44,067 goodware reports and 2,234 keylogger reports.

We performed an average length analysis of API sequences for different types of malware in the dataset, selecting the category with the smaller average length (Coinminer, with an average length of 311) as the reference. Subsequently, we conducted the performance tests within the range $L \in \{256, 512, 1,024, 2,048\}$ to identify the optimal sequence length. The performance peaked when the sequence length was $L = 1024$. Therefore, we set the sequence length to 1,024.

ML-based malware classifiers face significant challenges, including **sampling bias** [60] and **dataset shift** [61], [62]. In this work, since we evaluated the effectiveness of different feature engineering methods under the assumption of identical distributions between the training and testing data, the performance losses caused by distribution differences with real-world deployment data can be ignored. However, our results are affected by dataset shift, which necessitates extra caution when constructing the dataset. Dataset shifts can be broadly categorized into three types [63], and we implemented specific mitigations for each type:

1) Prior probability shift: This type of shift, called label shift, refers to changes in the label distribution $P(y \in Y)$, implying that the base proportion of a certain class changes over time. To address this shift, we followed the recommendations of previous studies [64], [65] to mitigate temporal and spatial biases. As shown in Fig. 4a, we re-split the training and test sets by month based on timestamps (i.e., the creation time of the malware). First, we select the family with the fewest samples each month as the reference, and perform random downsampling on the remaining families to ensure that the proportions of different malware families remain consistent each month. Then, based on the total number of malware in that month, we randomly downsample the goodware to maintain a 4:1 ratio of goodware to malware. Additionally, to avoid artificially boosting accuracy, we refrained from using k-fold cross-validation during training.

2) Covariate shift: Covariate shift refers to changes in the feature distribution $P(x \in X)$, where the frequency of certain features increases or decreases (e.g., changes in API call frequency over time). In the re-split dataset, we measured the frequency of four feature types within the first 1,024 API calls of each sample. Fig. 4b presents the cumulative feature proportion changes across all samples for each month, showing that the proportions of different feature types remain relatively

consistent across months.

3) Concept drift: Concept drift refers to changes in the conditional distribution $P(y \in Y \mid x \in X)$, occurring when the ground truth definition changes. For example, when new malware families emerge, the model may misclassify samples from these new families due to the limitations in prior knowledge, even if no covariate or prior probability shift has occurred. As shown in Fig. 4a, after splitting the dataset by time, we found that the Adload and Lokibot malware families appeared later in the timeline. Therefore, we excluded samples from these families in the training set, allowing them to appear exclusively in the test set as new families. This approach enables us to evaluate the model's performance under such concept drift conditions.

Table IV lists a detailed breakdown of the number of goodware samples and the sample counts for each malware family in the dataset.

B. Experimental Environment

The sandbox environment runs on a computer with Ubuntu 20.04 (64-bit), equipped with an Intel(R) Xeon(R) Gold 6230 CPU at 2.10 GHz, 256.0 GB of RAM, and a 1TB hard disk drive. The model training environment runs on a computer equipped with an NVIDIA GeForce RTX 4090 GPU (56GB VRAM), 64GB RAM, and an Intel(R) Core(TM) i9-13900 CPU (3.0GHz base frequency). The system was running a 64-bit version of Microsoft Windows 11 Professional. The development framework used for the experiments included Python 3.8.19, PyTorch 2.3.0, and CUDA 12.1. Additionally, other commonly used libraries, such as scikit-learn, numpy, sentencepiece, and seaborn, were employed. During the model training process, we employed the CrossEntropyLoss function and the AdamW optimizer. To assess the effectiveness of malware classification, we utilized metrics such as Accuracy(ACC), Recall(RC), Precision(PRE), and F1.

C. Hyperparameters

Throughout the learning process, it is common practice to generate different models by adjusting hyperparameters. To ensure a fair comparison, the best-performing model is typically selected and its performance on the test set is presented. Although this approach is generally reasonable, it can still suffer from a biased hyperparameter selection. Strict data isolation can effectively address the potential biases that may arise when determining hyperparameters and thresholds [27]. As shown in Fig. 4a, the dataset is split into training, validation, and test sets based on the timeline. For each hyperparameter (e.g., embedding vector dimension), we iteratively train each model within its predefined range, performing training and validation with different hyperparameters at each step, and ultimately selecting the set of hyperparameters that yields the best performance.

V. Evaluation Results

Table V presents the best results of each model with knowledge-based or NLP-based feature engineering. Models with knowledge-based feature engineering generally outperform those with NLP-based across all metrics. On the raw dataset, models with knowledge-based feature engineering achieve an F1 score approximately 20% higher than those with NLP-based. In the supplemented dataset, this gap narrows to around 10%. Notably, under the same conditions, the CNN model consistently outperforms the others, regardless of whether feature engineering is applied. This phenomenon may be attributed to the presence of significant pattern features in the API calls, particularly local API call patterns, CNN models, through convolution operations and the local receptive field mechanism, effectively capture these pattern features and directly fit classification boundaries in high-dimensional space, thereby enhancing their ability to model pattern information. In contrast, LSTM relies on sequential dependencies to capture long-range features, while the Transformer, despite possessing a global attention mechanism, may not perform as well as CNN in tasks requiring fine-grained local feature extraction due to the lack of an inherent local inductive bias.

> **Answer for RQ1**: Under the same conditions, models with knowledge-based feature engineering outperform those with NLP-based methods, and this effect is more pronounced when the sample size is small.

Figs. 5a and 5b display the classification confusion matrices on the raw dataset, using the CNN model with knowledge-based and NLP-based feature engineering, respectively. The classification accuracy for the "keylogger" category is nearly 0% in the model with NLP-based feature engineering. This outcome also explains why the CNN model, despite showing high ACC and PRE on the raw dataset, exhibits significantly lower RC and F1 scores. The primary reason for this is the insufficient number of samples in the "keylogger" category, which causes the model fail to effectively learn the characteristics of this category during the training, resulting in poor recognition performance during the testing. However, after applying knowledge-based feature engineering, the dataset shows marked improvement, especially in classifying minority classes, highlighting the significant effect of knowledge-based feature engineering in mitigating data imbalance.

Figs. 5d and 5e present the model's performance on the dataset supplemented with keylogger samples. Under NLP-based feature engineering, the classification accuracy for the keylogger category improves significantly, with the overall performance gains primarily stemming from the classification results for this category. However, despite this improvement, its performance remains inferior to the results achieved by knowledge-based feature engineering. Therefore, the model with knowledge-based feature engineering still outperforms the model with NLP-based feature engineering.

As shown in Figs. 5b and 5c, the Transformer model demonstrates greater sensitivity to keylogger samples with NLP-based feature engineering, outperforming the CNN model by successfully identifying some of these samples. However, as shown in Figs. 5e and 5f, after supplementing the dataset,

TABLE IV: Number of samples per malware family in Avast-CTU Dataset.

Family	**Goodware**	**Emotet**	**HarHar**	**njRAT**	**Qakbot**	**Swisyn**	**Trickbot**	**Ursnif**	**Zeus**	***Adload***	***Lokibot***
Samples	44,067	14,429	4,091	3,372	3,895	10,591	4,002	1,243	1,875	64	96

TABLE V: Results of different models on the raw and supplemented datasets.

Type	Model	Results on the raw dataset				Results on the supplemented dataset			
		ACC	PR	RC	F1	ACC	PR	RC	F1
Knowledge	**CNN**	**99.04**	**99.01**	**99.03**	**99.02**	**99.13**	**99.13**	**99.12**	**99.13**
	LSTM	98.67	98.63	98.67	98.65	98.61	98.64	98.61	98.15
	Transformer	97.00	96.91	97.00	96.93	97.77	97.41	96.82	96.96
NLP	**CNN**	**95.15**	**93.29**	**76.64**	**77.46**	**94.80**	**90.82**	**89.08**	**89.69**
	LSTM	94.09	91.97	75.70	76.15	93.13	86.99	87.83	87.39
	Transformer	93.77	90.01	75.40	75.45	92.46	86.43	85.27	85.76

the performance increase of the Transformer model in this category is limited and does not align with the performance of the CNN model.

It is noteworthy that all models tend to misclassify keylogger samples as RAT type. This is because RATs typically include keyboard input capture functionality, which overlaps with the behavioral characteristics of keyloggers, increasing the difficulty for the model to distinguish between the two. Moreover, RATs exhibit more diverse and complex behaviors than keyloggers, often incorporating actions such as network communication and privilege escalation, in addition to keyboard capture. These rich features enable the model to more accurately identify RATs, thereby reducing the likelihood of misclassifying RAT samples as keyloggers.

A. Ablation Study

To highlight the utility of various features in malware classification, we examined the impact of different combinations of four types of features on the final performance of the model. For the ablation experiment design, we used the supplementary dataset, as it offers a broad representation of behaviors. In addition to using only APIs and only parameters as input features, we also included combinations such as API with virtual address parameters (API + address), API with string parameters (API + string), API with integer parameters (API + integer), and all features together, yielding a total of six combination modes. The performance results are represented using ROC curves, where the True Positive Rate (TPR) indicates the proportion of correctly identified instances of a given class, and the False Positive Rate (FPR) represents the proportion of other classes misclassified as that class. The final ROC curve is generated by averaging the TPR and FPR values across all classes.

As shown in Fig. 6a, feature reduction has varying degrees of impact on the model's performance under knowledge-based feature engineering. The performance declines when any single feature is used, compared to using all features. Notably, the combination of API with string performs similarly to using only the API feature, while the worst performance is observed when only parameters are used. However, as shown in Fig. 6b, under NLP-based feature engineering, using only API features achieves the best performance, the introduction of any type of parameter feature results in inferior performance compared to using only API features, with the API + string combination resulting in the lowest performance. We hypothesize that the introduction of numerous string parameters may make it difficult to distinguish whether the input string represents an API name or a parameter, thus disrupting the API patterns in the call sequences. Regardless of the use of feature engineering, the results consistently highlight the critical role of APIs in malware classification.

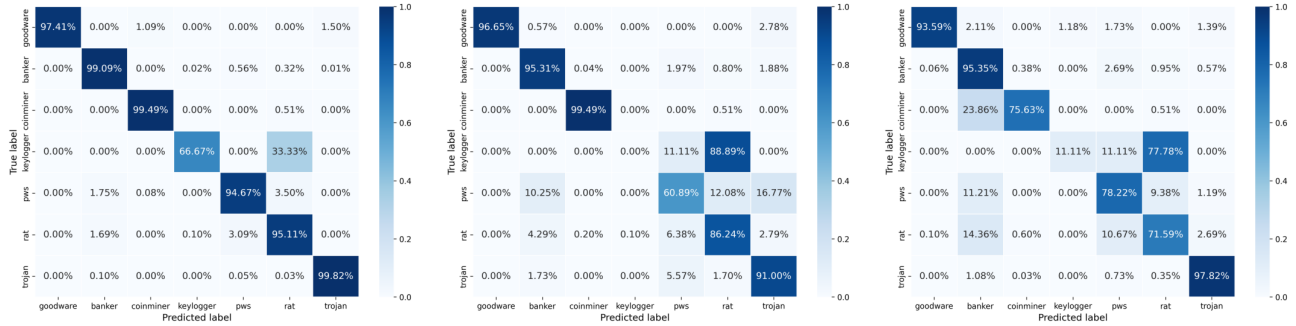
Answer for RQ2: Regardless of what type of feature engineering is applied, API features consistently contribute the most to the classification results. After applying knowledge-based feature engineering, incorporating any type of parameter feature leads to an improvement in the model's performance. However, in the NLP-based feature engineering, the performance of the model with any type of parameter features is inferior to that of the model using only API features.

VI. Key Factor Assessment

In this section, we employ XAI techniques to assess the key factors behind the model's classification, and further examine artifacts unrelated to a model. In other words, which features of the API sequence contribute the most to the model's classification decision. Based on prior research, a target model can be treated as a black box [66], [67]. Using simple models to approximate the decision boundaries near the input samples, we identify key features for generating explanations. Moreover, we apply gradient-based methods, utilizing backpropagation in deep neural networks to measure the sensitivity of each feature [68], [69]. For Transformer models, we use the attention activation values in the encoder layers to evaluate the relative importance of each token, providing interpretability analysis.

A. Knowledge-based Method

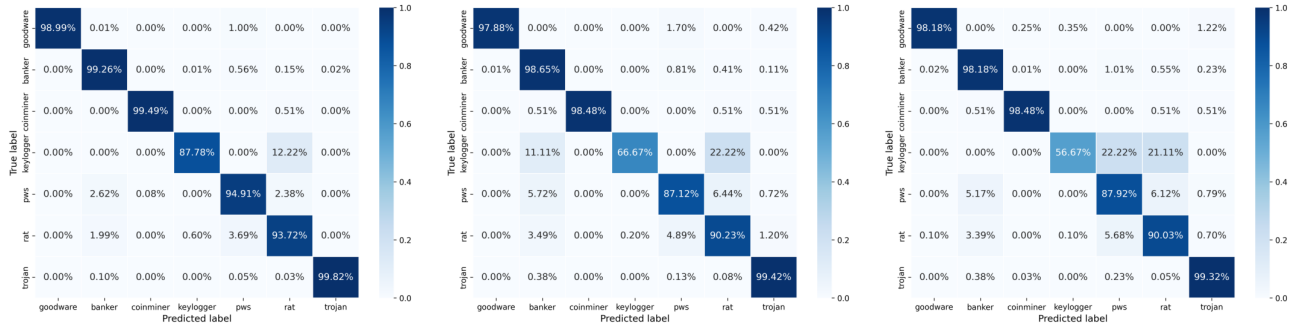
Given the use of expert-designed heuristic rules in feature engineering, the model is trained to learn key features.



(a) Classification confusion matrix on the raw dataset using the CNN with knowledge-based feature engineering.

(b) Classification confusion matrix on the raw dataset using the CNN with NLP-based feature engineering.

(c) Classification confusion matrix on the raw dataset using the Transformer with NLP-based feature engineering.



(d) Classification confusion matrix on the supplementary dataset using the CNN with knowledge-based feature engineering.

(e) Classification confusion matrix on the supplementary dataset using the CNN with NLP-based feature engineering.

(f) Classification confusion matrix on the supplementary dataset using the Transformer with NLP-based feature engineering.

Fig. 5: Classification confusion matrices using different processing methods on the raw and supplemented datasets.

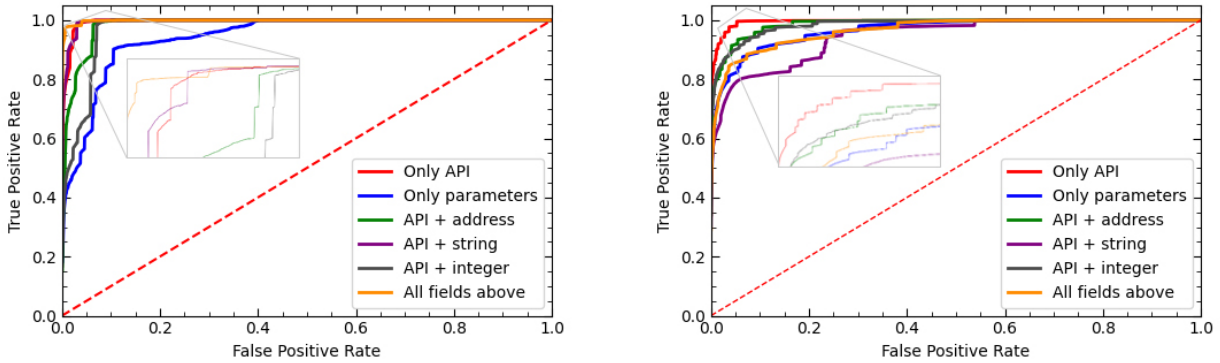
Pearson's correlation coefficient can be used to assess the linear relationship between the extracted features and different malware families. This method does not require understanding the internal structure of the model but rather evaluates the relationship by analyzing the correlation between the features and the output. As shown in Table VII, the contribution of features varies across different categories, string parameters and address parameters exhibit correlations ($r > 0.6$) in distinguishing certain specific categories, such as PWS and goodware, suggesting that these features have a significant impact on predicting the categories and may serve as crucial indicators in the classification process. By contrast, integer parameters are relatively effective in distinguishing goodware, but their overall correlation is low, implying that relying solely on this feature may not accurately differentiate between categories. Notably, virtual addresses exhibit a surprisingly correlation overall.

B. NLP-based Method

In contrast to the manual heuristic rules of knowledge-based feature engineering, the application of NLP methods enables the model to autonomously learn and capture the behavioral features or heuristic patterns. SHAP [69] is employed to compute the Shapley values of individual tokens, facilitating the differentiation of each feature's relevance within the malware classification. The token contributions derived here incorporate both positive and negative effects, calculating the cumulative sum of the absolute SHAP values for each token.

As shown in Table VI, in comparison to goodware, most malware types share certain key tokens, such as `modulehandle`, `dll`, and `functionaddress`. These tokens represent fundamental system-level APIs or resource identifiers, which are commonly found across various types of malware. This commonality indicates that underlying system resources are prevalent features in malicious activities. It is noteworthy that the model often relies on handles and virtual addresses in classification tasks, but they are difficult for humans to interpret and vary across different executions of the same binary. This makes them unsuitable as stable indicative features for classification. Methods such as feature hashing [29] are needed to improve their generalizability, which highlights the importance of manual feature engineering.

Time-related tokens, such as `milliseconds` and `NtDelayExecution`, exhibit significant weights across various malware types, particularly within the RAT category. This suggests that malware commonly utilizes timing-based scheduling and delay execution strategies to evade detection, thereby enhancing the stealthiness of malicious activities.



(a) Results of Knowledge-based feature engineering.

(b) Results of NLP-based feature engineering.

Fig. 6: The impact of different feature combinations on the model's final performance.

TABLE VI: The top 10 individual tokens influencing the model's decision across different malware types.

Types	Top 10 individual Tokens (SHAP Value)
Goodware	hkey, lpsubkey, regopenkeyexw, local, machine, software, microsoft, hfile, 1, generic
Banker	**modulehandle**, 0, ldrgetprocedureaddress, ordinal, **functionaddress**, **dll**, 0x00000000, kernel32, baseaddress, 0xffffffff
Coinminer	kernel32, disableusermodecallbackfilter, **dll**, getnumberofconsoleinputevents, **modulehandle**, **functionaddress**, ldrgetprocedureaddress, 0xfffffffe, 0, getsystemtimeasfiletime
Keylogger	**functionaddress**, regqueryvalueexw, ordinal, ntdelayexecution, **modulehandle**, **dll**, milliseconds, kernel32, 0xffffffff, stackpivoted
PWS	disableusermodecallbackfilter, `<unk>`, **modulehandle**, **dll**, oleaut32, 0x00000000, getsystemtimeasfiletime, **functionaddress**, ldrgetprocedureaddress, milliseconds
RAT	disableusermodecallbackfilter, getsystemtimeasfiletime, **dll**, **modulehandle**, 0, **functionaddress**, ldrgetprocedureaddress, ntdelayexecution, milliseconds, pi
Trojan	disableusermodecallbackfilter, 0x00006000, **dll**, getsystemtimeasfiletime, oleaut32, 0, milliseconds, **modulehandle**, ordinal, **functionaddress**

C. The Attention Mechanism

For the Transformer model, attention activations can be used to indicate the relative importance of different tokens within the model. We present an analysis of a sample classified as Trojan (MD5: *227958c8e6e50ac28ffeb146156e82a5*), focusing on the biases in the attention weights and their implications.

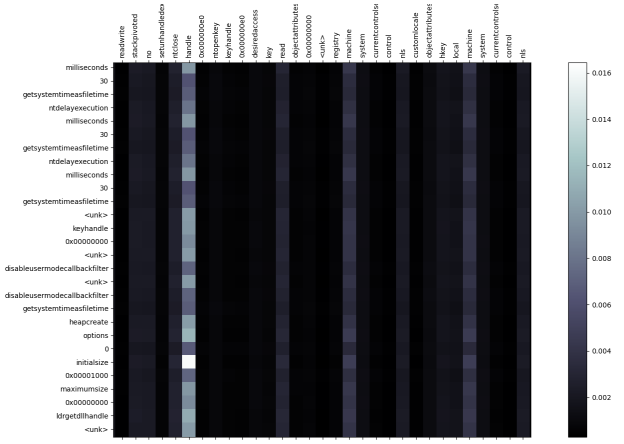


Fig. 7: Attention activations for different tokens in the first attention layer of the model.

In the knowledge-based feature engineering approach, the model particularly highlights two segments of function calls in the sample sequence: between tokens 9-13 and 92-98, as shown in Fig. 8. In the first group of API calls, `NtQueryValueKey` is a system call used for querying Windows registry values. The model focuses specifically on the query of the registry key `DisableUserModeCallbackFilter`, which is commonly associated with the anti-debugging or virtual machine detection. Additionally, `LdrGetDllHandle` is used to retrieve the handle of a loaded DLL, and `LdrGetProcedureAddress` is used to retrieve the address of an exported function from a DLL. These API calls are commonly involved in dynamically loading libraries and accessing specific functions, which may be a part of the process of setting up or interacting with system components, and potentially installing system hooks to capture user operations.

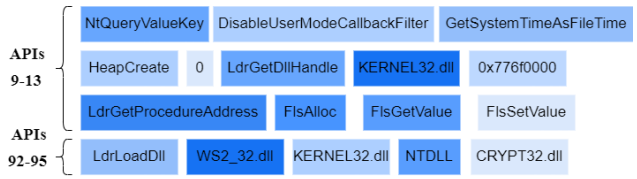


Fig. 8: Highlight the areas with the highest attention in the sample under knowledge-based feature engineering processing, where darker colors indicate greater attention.

As shown in Fig. 7, by contrast, the model with NLP-based feature engineering primarily focuses on APIs related to system time retrieval (`GetSystemTimeAsFileTime`), delayed execution (`NtDelayExecution`), and registry access (`keyhandle`). These actions mainly reflect the anti-debugging behaviors; in particular, the delayed execution is typically used to outlast the analysis periods, assuming that the analysis lasts only for a limited time. However, these anti-debugging and anti-sandbox techniques are commonly used in malware and are not those core characteristics of Trojans. Clearly, their importance is much less than that of behaviors, such as installing system hooks, which are emphasized under feature engineering. Notably, both models focus on the `GetSystemTimeAsFileTime` system call, which retrieves and returns the current system time as a file time in the timestamp format. This function is commonly used for logging or event time synchronization. Malware may leverage this to mark the time of its operations or synchronize events during the execution of malicious activities.

Answer for RQ3: A model typically relies on API names, handles, and library addresses to distinguish between different types of malware. Models with NLP-based feature engineering tend to focus more on superficial features, such as anti-debugging behaviors, while models with knowledge-based feature engineering are more effective at capturing the core characteristics of samples (e.g., the malicious behaviors like installing system hooks).

VII. Discussion

1) Threat to validity: Like all empirical studies, our observations are subject to some threats to the validity. In the following, we discuss the potential sources of bias and how we address them with our best-efforts.

To ensure that those variations in model performance are solely attributed to different feature engineering approaches rather than other external factors, we maintain consistency in data sources and preserve uniformity in computational methods and evaluation criteria. To mitigate the selection bias, we carefully design inclusion and exclusion criteria for the dataset to achieve a sample balance. Additionally, to reduce the impact of the model selection bias, we conduct our experiments using three models with distinct characteristics, thereby mitigating the result deviations caused by the differences in model architectures. Regarding the external validity threats, particularly the generalizability of our findings, we acknowledge that the applicability of this study is constrained by the analyzed data sources. Given the widespread adoption of API analysis in the security research, our work primarily focuses on different feature engineering techniques for API call sequences. We believe our findings hold a certain degree of representativeness in this domain. However, since model performance may vary with the changes in the dataset characteristics, a careful evaluation is required when applying our conclusions to other data environments.

TABLE VII: Pearson correlation coefficients of features.

Type	API Name	API Integer	API String	API Address
Goodware	0.295	0.373	**-0.726**	**-0.483**
Banker	-0.202	-0.341	0.249	0.143
Coinminer	-0.39	0.018	-0.042	**-0.582**
Keylogger	0.213	-0.009	0.192	0.015
PWS	0.06	0.096	**0.631**	**-0.422**
RAT	0.349	-0.053	0.188	**0.482**
Trojan	0.136	-0.057	0.417	0.339
All	**0.4**	-0.091	0.396	**0.512**

2) Recommendations: The models using knowledge-based feature engineering outperform the models using NLP-based methods. Due to the disparity between the features utilized by ML models and those selected by human analysts, it is necessary to reassess the potential features that are often overlooked due to their interpretability challenges for humans. When introducing new features, caution must be exercised in organizing the feature inputs to avoid interference or redundancy among features.

API name features play a critical role in malware classification. Given the significant pattern features present in API calls, especially local API call patterns, the model design should prioritize the architectures such as CNN that are more focused on capturing local features, to improve classification accuracy.

Models often rely on fixed-value features in classification tasks, such as handles and address values. Such a dependency may limit the model's generalization ability, as it could become overly dependent on these fixed-value features, and thus reducing its capability to identify broader patterns of malicious behaviors. These features require manual feature engineering to be effectively utilized.

3) Limitation: When being executed in virtualized or emulated environments, some malware may avoid exposing their malicious behaviors due to the intervention of environment-aware detection mechanisms [70]. The dataset we used does not account for such a situation. Malware that focuses on sandbox evasion techniques [71] does not produce high-quality execution reports, making it difficult to determine whether it is malware. This highlights the need for techniques to counter evasion tactics in production environments.

VIII. Conclusion

In this paper, we provide new insights into the impact of feature engineering upon malware classification. We used

the complete set of data features from API call sequences, applying both knowledge-based and NLP-based feature engineering methods and evaluated their performance under three ML models. Our results show that knowledge-based feature engineering methods consistently outperform NLP-based ones, particularly when dealing with small sample datasets. Regardless of the feature engineering method being applied, CNN consistently achieves the best performance among three ML models. In addition, there is a stark contrast in the performance of knowledge-based and NLP-based methods when increasing the number of input features. Our analysis further reveals that models often focus on features such as handles and virtual address values, which are challenging for humans to interpret and also execution-environment dependent.

IX. Acknowledgements

We thank the anonymous reviewers for their insightful and constructive feedback. This work was supported by the Industrial Foundation Reconstruction and High-Quality Development of Manufacturing Industry Special Project (0747-2361SCCZA193). Zhi Li is the corresponding author.

References

[1] AV-TEST Institute, "Malware statistics & trends," 2024, accessed: 2024-10-28. [Online]. Available: https://www.av-test.org/en/statistics/malware/

[2] S. Dambra, Y. Han, S. Aonzo, P. Kotzias, A. Vitale, J. Caballero, D. Balzarotti, and L. Bilge, "Decoding the secrets of machine learning in malware classification: A deep dive into datasets, feature extraction, and model performance," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023, pp. 60–74.

[3] G. Shenderovitz and N. Nissim, "Bon-apt: Detection, attribution, and explainability of apt malware using temporal segmentation of api calls," *Computers & Security*, vol. 142, p. 103862, 2024.

[4] X. Deng, H. Tang, X. Pei, D. Li, and K. Xue, "Mdhe: A malware detection system based on trust hybrid user-edge evaluation in iot network," *IEEE Transactions on Information Forensics and Security (TIFS)*, 2023.

[5] J. Jeon, B. Jeong, S. Baek, and Y.-S. Jeong, "Static multi feature-based malware detection using multi spp-net in smart iot environments," *IEEE Transactions on Information Forensics and Security (TIFS)*, 2024.

[6] Š. Balogh and J. Mojžiš, "New direction for malware detection using system features," in *10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 1, 2019, pp. 176–183.

[7] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, "UNVEIL: A Large-Scale, automated approach to detecting ransomware," in *25th USENIX Security Symposium (USENIX Security)*, 2016, pp. 757–772.

[8] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, X. Wang *et al.*, "Effective and efficient malware detection at the end host." in *18th USENIX Security Symposium (USENIX Security)*, vol. 4, 2009, pp. 351–366.

[9] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[10] D. Trizna, L. Demetrio, B. Biggio, and F. Roli, "Nebula: Self-attention for dynamic malware analysis," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 19, pp. 6155–6167, 2024.

[11] C. Jindal, C. Salls, H. Aghakhani, K. Long, C. Kruegel, and G. Vigna, "Neurlux: dynamic malware analysis without feature engineering," in *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, 2019, pp. 444–455.

[12] Z. Zhang, P. Qi, and W. Wang, "Dynamic malware analysis with feature engineering and feature learning," in *Proceedings of the AAAI conference on artificial intelligence (AAAI)*, vol. 34, 2020, pp. 1210–1217.

[13] S. Aonzo, Y. Han, A. Mantovani, and D. Balzarotti, "Humans vs. machines in malware classification," in *32th USENIX Security Symposium (USENIX Security)*, 2023, pp. 1145–1162.

[14] M. Yong Wong, M. Landen, M. Antonakakis, D. M. Blough, E. M. Redmiles, and M. Ahamad, "An inside look into the practice of malware analysis," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021, pp. 3053–3069.

[15] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Advances in Artificial Intelligence*. Springer, 2016, pp. 137–149.

[16] H. Zhang, W. Zhang, Z. Lv, A. K. Sangaiah, T. Huang, and N. Chilamkurti, "Maldc: a depth detection method for malware based on behavior chains," *World Wide Web*, vol. 23, pp. 991–1010, 2020.

[17] R. Agrawal, J. W. Stokes, M. Marinescu, and K. Selvaraj, "Neural sequential malware detection with parameters," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018, pp. 2656–2660.

[18] Z. Salehi, A. Sami, and M. Ghiasi, "Maar: Robust features to detect malicious activity based on api calls, their arguments and return values," *Engineering Applications of Artificial Intelligence*, vol. 59, pp. 93–102, 2017.

[19] C. Li, Z. Cheng, H. Zhu, L. Wang, Q. Lv, Y. Wang, N. Li, and D. Sun, "Dmalnet: Dynamic malware analysis based on api feature engineering and graph learning," *Computers & Security*, vol. 122, p. 102872, 2022.

[20] T. Chen, H. Zeng, M. Lv, and T. Zhu, "Ctimd: Cyber threat intelligence enhanced malware detection using api call sequences with parameters," *Computers & Security*, vol. 136, p. 103518, 2024.

[21] X. Chen, Z. Hao, L. Li, L. Cui, Y. Zhu, Z. Ding, and Y. Liu, "Cruparamer: Learning on parameter-augmented api sequences for malware detection," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 17, pp. 788–803, 2022.

[22] C. Liu, B. Li, J. Zhao, W. Feng, X. Liu, and C. Li, "A2-clm: Few-shot malware detection based on adversarial heterogeneous graph augmentation," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 19, pp. 2023–2038, 2024.

[23] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici, "Generic black-box end-to-end attack against state of the art api call based malware classifiers," in *Research in Attacks, Intrusions, and Defenses (RAID)*, M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds., 2018, pp. 490–510.

[24] R. Tian, R. Islam, L. Batten, and S. Versteeg, "Differentiating malware from cleanware using behavioural analysis," in *5th International Conference on Malicious and Unwanted Software*, 2010, pp. 23–30.

[25] E. B. Karbab and M. Debbabi, "Maldy: Portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports," *Digital Investigation*, vol. 28, 2019.

[26] C. Liu, B. Li, J. Zhao, X. Liu, and C. Li, "Malaf: Malware attack foretelling from run-time behavior graph sequence," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2023.

[27] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and don'ts of machine learning in computer security," in *31th USENIX Security Symposium (USENIX Security)*, 2022, pp. 3971–3988.

[28] B. Bosansky, D. Kouba, O. Manhal, T. Sick, V. Lisy, J. Kroustek, and P. Somol, "Avast-ctu public cape dataset," 2022. [Online]. Available: https://arxiv.org/abs/2209.03188

[29] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, 2009, pp. 1113–1120.

[30] M. R. Smith, N. T. Johnson, J. B. Ingram, A. J. Carbajal, B. I. Haus, E. Domschot, R. Ramyaa, C. C. Lamb, S. J. Verzi, and W. P. Kegelmeyer, "Mind the gap: On bridging the semantic gap between machine learning and malware analysis," in *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, 2020, pp. 49–60.

[31] N. Peiravian and X. Zhu, "Machine learning for android malware detection using permission and api calls," in *IEEE 25th International Conference on Tools with Artificial Intelligence*, 2013, pp. 300–305.

[32] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze, "Malware detection based on mining api calls," in *Proceed-*

ings of the ACM Symposium on Applied Computing, 2010, pp. 1020–1025.
[33] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, and S. Mukkamala, "Malware detection using assembly and api call sequences," *Journal in Computer Virology*, vol. 7, pp. 107–119, 2011.
[34] https://learn.microsoft.com/en-us/windows/win32/api/, accessed: 2024-12-25.
[35] X.-w. Chen and J. C. Jeong, "Enhanced recursive feature elimination," in *IEEE Sixth International Conference on Machine Learning and Applications (ICMLA)*, 2007, pp. 429–435.
[36] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, pp. 433–459, 2010.
[37] S. Zhang, J. Wu, M. Zhang, and W. Yang, "Dynamic malware analysis based on api sequence semantic fusion," *Applied Sciences*, vol. 13, p. 6526, 2023.
[38] R. Islam, R. Tian, L. Batten, and S. Versteeg, "Classification of malware based on string and function feature selection," in *IEEE Second Cybercrime and Trustworthy Computing Workshop*, 2010, pp. 9–17.
[39] R. Islam, R. Tian, L. M. Batten, and S. Versteeg, "Classification of malware based on integrated static and dynamic features," *Journal of Network and Computer Applications*, vol. 36, pp. 646–656, 2013.
[40] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq, "Using spatio-temporal information in api calls with machine learning algorithms for malware detection," in *Proceedings of the 2th ACM Workshop on Security and Artificial Intelligence*, 2009, pp. 55–62.
[41] E. M. Rudd, F. N. Ducau, C. Wild, K. Berlin, and R. Harang, "ALOHA: Auxiliary loss optimization for hypothesis augmentation," in *28th USENIX Security Symposium (USENIX Security)*, 2019, pp. 303–320.
[42] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.
[43] R. Sennrich, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.
[44] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Network and Distributed Systems Security Symposium (NDSS)*, vol. 14, 2014, pp. 23–26.
[45] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "Droidevolver: Self-evolving android malware detection system," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019, pp. 47–62.
[46] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "Droidsieve: Fast and accurate classification of obfuscated android malware," in *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 309–320.
[47] https://www.kaggle.com/datasets/blackarcher/malware-dataset, accessed: 2024-09-22.
[48] https://www.kaggle.com/competitions/malware-classification/data, accessed: 2024-09-22.
[49] https://zenodo.org/record/1203289, accessed: 2024-09-22.
[50] H. S. Anderson and P. Roth, "Ember: an open dataset for training static pe malware machine learning models," *arXiv preprint arXiv:1804.04637*, 2018.
[51] F. O. Catak and A. F. Yazı, "A benchmark api call dataset for windows pe malware classification," *arXiv preprint arXiv:1905.01999*, 2019.
[52] G. Severi, T. Leek, and B. Dolan-Gavitt, "Malrec: compact full-trace malware recording for retrospective deep analysis," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2018, pp. 3–23.
[53] A. Brukhovetskyy and K. O'Reilly, "Cape sandbox v2.1 book," 2022, accessed on 08/27/2024. [Online]. Available: https://capev2.readthedocs.io/en/latest/index.html
[54] portableapps.com, accessed: 2024-09-12.
[55] sourceforge.net, accessed: 2024-09-12.
[56] softonic.com, accessed: 2024-09-12.
[57] https://www.virustotal.com/gui/home/upload, accessed: 2024-07-22.
[58] https://virusshare.com/, accessed: 2024-07-29.
[59] A. Küchler, A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti, "Does every second count? time-based evolution of malware behavior in sandboxes," in *Network and Distributed Systems Security Symposium (NDSS)*, 2021.
[60] S. Thirumuruganathan, F. Deniz, I. Khalil, T. Yu, M. Nabeel, and M. Ouzzani, "Detecting and mitigating sampling bias in cybersecurity with unlabeled data," in *33th USENIX Security Symposium (USENIX Security)*, 2024, pp. 1741–1758.
[61] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting concept drift in malware classification models," in *26th USENIX Security Symposium (USENIX Security)*, 2017, pp. 625–642.
[62] B. Miller, A. Kantchelian, M. C. Tschantz, S. Afroz, R. Bachwani, R. Faizullabhoy, L. Huang, V. Shankar, T. Wu, G. Yiu *et al.*, "Reviewer integration and performance measurement for malware detection," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016, pp. 122–141.
[63] J. G. Moreno-Torres, T. Raeder, R. Alaiz-Rodríguez, N. V. Chawla, and F. Herrera, "A unifying view on dataset shift in classification," *Pattern Recognition*, vol. 45, pp. 521–530, 2012.
[64] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *28th USENIX Security Symposium (USENIX Security)*, 2019, pp. 729–746.
[65] M. Mimura, "Impact of benign sample size on binary classification accuracy," *Expert Systems with Applications*, vol. 211, p. 118630, 2023.
[66] M. T. Ribeiro, S. Singh, and C. Guestrin, "" why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 1135–1144.
[67] ——, "Anchors: High-precision model-agnostic explanations," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, vol. 32, 2018.
[68] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 618–626.
[69] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*, 2017, p. 4768–4777.
[70] D. Li and Q. Li, "Adversarial deep ensemble: Evasion attacks and defenses for malware detection," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 15, pp. 3886–3900, 2020.
[71] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting environment-sensitive malware," in *Recent Advances in Intrusion Detection (RAID)*, 2011, pp. 338–357.

APPENDIX

The table below shows the optimal hyperparameters for different models to reproduction.

TABLE VIII: Key Hyperparameters and Optimal Values for the Models

Model	Hyperparameter	Optimal Value	
		Knowledge	NLP
CNN	Vector dimensions	132	96
	k-gram	2, 3, 4, 5	2, 3, 4, 5
	Kernel channel	128	128
	Dropout rate	0.3	0.3
LSTM	Vector dimensions	132	64
	LSTM hidden units	64	256
	LSTM layers	2	1
	LSTM dropout	0.1	0.1
Transformer	Vector dimensions	132	64
	Number of attention heads	8	8
	Hidden layer dimensions	256	256
	Number of encoder layers	2	2
	Dropout	0.2	0.3
MLP	Units of hidden layers	(128, 64)	(128, 64)
	Dropout rate	0.2	0.2