Results in Engineering 25 (2025) 104250

Contents lists available at ScienceDirect

Results in Engineering

journal homepage: www.sciencedirect.com/journal/results-in-engineering

Research paper

Transformer-based malware detection using process resource utilization metrics

Dimosthenis Natsos [a,b,*], Andreas L. Symeonidis [a,b]

[a] School of Electrical and Computer Engineering, AUTH, Thessaloniki, 54124, Greece
[b] Cyclopt PC, Thessaloniki, 55535, Greece

ARTICLE INFO

Keywords:
Dynamic malware signatures
Malware cascading effect
Malware detection
Multivariate series classification
Performance metrics
Resource utilization metrics
XAI

ABSTRACT

Malware detection has long relied on signature-based methods limited in detecting zero-day malware attacks. Although efficient, these approaches are vulnerable to obfuscation and evasion techniques. To this end, dynamic approaches utilizing process resource-utilization metrics have emerged as promising alternatives. They solve the aforementioned issues, but require large datasets for training and struggle with false-positives and false-negatives. This study is the first to explore the application of Transformers for malware detection using process resource-utilization metrics, encoding input data as sequences of processes, with each process represented by its resource-utilization metrics (e.g., CPU, memory, and disk usage). We compare the proposed Transformer-based architecture with the leading LSTM model in terms of accuracy, precision, recall, F1-score and training time, focusing on performance across varying sample sizes and validate our results with rigorous statistical methodologies. Our findings demonstrate Transformers' ability to maintain high performance even with smaller datasets, thus excel in real-world scenarios of limited data availability, and scale effectively with larger datasets, offering lower false-positive and false-negative rates. We shed light on the models' decision-making processes, introducing the concept of dynamic malware signatures derived from resource-utilization metrics and identifying key features that prominently reflect malware activity. Additionally, we showcase that other tenant processes within the operating system act as indirect indicators of malware presence, providing valuable signals for detection even when the malware process itself is not directly observed. This work establishes Transformers as the state-of-the-art solution for malware detection using process resource-utilization metrics, offering improved accuracy, scalability, and robustness over existing methods.

1. Introduction

Malicious software (malware) is a computer program that is inserted into a system, usually covertly, with the intent of compromising the victim's data, applications, or operating system (OS) or of otherwise annoying or disrupting the victim [1]. The scientific community has been fighting malware for over two decades. However, the automated detection of malicious software remains an open ended problem, due to the ever-evolving nature of the threats, in addition to the constant digitization of information, resources, processes and human transactions. The era of Artificial Intelligence (AI), Cloud Computing and Internet of Things (IoT) has accelerated the interconnection and smartification of computational resources, and has perpetually increased the attack vector for malware to exploit and misuse devices, systems and sensitive information. These advancements take place across a wide variety of domains, including smart agriculture systems [2], smart vehicles [3], smart manufacturing systems [4], and smart cities [5], among others. Despite these technological strides, many of these systems lack robust cybersecurity measures to protect both the infrastructures and the vast amounts of data they generate and process [5–8]. Furthermore, modern computing environments, characterized by complex software interactions and the rapid evolution of malware employing sophisticated evasion and obfuscation techniques, pose significant challenges for traditional detection methods. These limitations increase the risk of failing to detect novel or zero-day malware, threatening individuals, organizations, and critical infrastructure with potential consequences such as data breaches, financial losses, service disruptions, and even physical damage [9,10]. This vulnerability underscores the urgent need for advanced security mechanisms capable of efficiently detecting and countering malware. For this reason, the research community is exploring novel ways for utilizing AI,

* Corresponding author at: School of Electrical and Computer Engineering, AUTH, Thessaloniki, 54124, Greece.
E-mail addresses: dcnatsos@ece.auth.gr (D. Natsos), symeonid@ece.auth.gr (A.L. Symeonidis).

https://doi.org/10.1016/j.rineng.2025.104250
Received 25 November 2024; Received in revised form 31 January 2025; Accepted 2 February 2025

Available online 5 February 2025

Machine Learning (ML) and Deep Learning (DL) techniques for effectively detecting malware and protecting the cyber-physical space and it's users from malicious actors.

Malware detection using machine learning is usually categorized in static and dynamic (behavioral) malware detection [11]. Static malware detection involves analyzing code features, such as byte sequences [12], opcodes [13], n-Grams [14] and Portable Executable Headers [15], among others, that can be identified without executing the software. These features can be extracted directly from the program's binary file, making static analysis relatively fast and efficient. However, static analysis is susceptible to code obfuscation and packing techniques employed by malware authors to evade detection. Sophisticated malware can morph its code structure or encrypt its payload, rendering traditional static analysis ineffective [9,10]. This limitation has driven the search for complementary methods that can detect previously unseen or obfuscated malware.

Dynamic malware detection, on the other hand, analyzes the behavior of programs and processes during their execution to identify malicious activities. Unlike static analysis, which provides a snapshot of the program's code, dynamic analysis focuses on the real-time behavior of software, capturing runtime features such as API calls [16], network traffic data [17], hardware performance counters analytics [18] and memory access patterns [19]. These features allow dynamic analysis to identify malware based on how it interacts with the system, making it more robust against evasion techniques that static analysis cannot handle. For instance, malware can hide its malicious code but cannot completely mask behavioral anomalies like frequent system calls to unauthorized servers or unexpected spikes in memory usage. These behaviors are detectable through dynamic analysis.

Within dynamic analysis, the choice of features plays a crucial role in detection accuracy and efficiency. While low-level features like API calls, system calls and memory dumps provide a granular view of program behavior, they often require deeper system introspection, often through forensic tools, reverse engineering, or specialized monitoring software for extraction. This paper focuses on a different category of dynamic features: high-level process performance metrics (also known as process resource utilization metrics). These features include metrics such as CPU usage, memory consumption, disk activity, and network Input/Output (I/O), among others. Unlike low-level features, resource utilization metrics have the potential to capture the cascading effects of malware across the operating system, offering a holistic view of system behavior. For example, malware may create irregular CPU usage patterns or increase I/O operations in a way that disrupts normal system processes. Furthermore, resource utilization metrics are readily available through standard system monitoring tools and can be readily extracted across different operating systems, making them easily accessible for security applications. At the same time, they are less susceptible to evasion techniques that target specific low-level features [9,10]. These high-level indicators are particularly valuable in detecting novel or zero-day malware that may evade traditional signature-based detection methods, as they capture the broader impact of malicious activity on system resources, regardless of the specific code used.

In both static and dynamic approaches, machine learning algorithms are trained on known malware and benign samples to create models that can potentially classify new, unseen programs as malicious or benign. A plethora of ML and DL models and techniques has been proposed (supervised, unsupervised, semi-supervised and reinforcement learning), usually employing models such as Decision Trees, Support Vector Machines, Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), among others [20]. These models have demonstrated varying degrees of success in malware detection, but they are often limited by their dependence on large, labeled datasets and their inability to effectively model long-term dependencies in sequential data. Recently, the introduction of Transformers has revolutionized the Natural Language Processing domain [21]. Initially developed for tasks like machine translation, Transformers have demonstrated exceptional capabilities in capturing long-range dependencies and contextual relationships in data, making them highly effective in real-world applications. Their application in malware detection, particularly in leveraging dynamic features such as resource utilization metrics, represents a promising realm in the fight against increasingly sophisticated cyber threats. Nevertheless, the literature has yet to explore the application of Transformers in the domain of malware detection using process resource utilization metrics.

This work presents a novel approach for malware detection, leveraging the power of Transformers to analyze high-level process performance metrics. This is the first study, to the best of our knowledge, to explore the application of Transformers in this specific context. Our system employs resource utilization metrics of system processes to classify them as benign or malicious. We make use of the resource utilization dataset published in [22] and compare the performance and training time of our proposed architecture with the so-far leading model on this problem, the Long Short-Term Memory (LSTM) RNN. We explore the potential of the proposed architecture in variable size samples and assess its performance and robustness on smaller, representative datasets which is usually the case in real-world scenarios. Ultimately, we validate our results using established statistical methods. Our results reveal that the proposed Transformer architecture consistently outperforms the state-of-the-art LSTM approach, excelling with both smaller and larger datasets. This underscores the Transformer's superior performance, scalability, and robustness across varying dataset sizes. Additionally, we examine the training times of the two models, demonstrating that, despite their complexity, Transformers become faster to train as the volume of data increases. This observation aligns with existing findings on the superior performance of Transformers compared to traditional Deep Neural Network (DNN) architectures [23,21]. Furthermore, we focus on the explainability of our approach by answering the question: *how can these models detect malware so effectively, using only resource utilization metrics?* We apply three different feature attribution methods, in order to study the importance of the resource utilization features in the decision-making process of the proposed architecture. In this way, we introduce the concept of dynamic malware signatures derived from process resource utilization metrics, identifying features that prominently reflect malware activity for detection purposes. Finally, we showcase the cascading effects of malware behavior on system-wide processes and evaluate the role of the malicious process itself in achieving detection through the proposed architecture. The results indicate strong potential for detecting malware by examining indirect signs of infection across the system, rather than focusing solely on direct indicators (such as the malware process itself). This implies that malware leaves dynamic imprints on other tenant OS processes, providing valuable clues for detection.

The main contributions of our work are the following:

- We propose a novel Transformer-based architecture for detecting malware using process resource utilization metrics. This architecture is compared against the leading LSTM model, demonstrating its superior performance and highlighting the potential of Transformer models in malware detection.
- We conduct a detailed comparative analysis of the Transformer and LSTM models across varying sample sizes. Our results demonstrate the ability of Transformers to maintain high performance with smaller datasets, proving more suitable under conditions of limited data availability or when reduced training times are required, and scale effectively with larger ones, thus making them the superior choice for real-word scenarios.
- We shed light to the black-box nature of Transformers using explainable methodologies, and provide an overall understanding of how the proposed solution makes classification choices, as well as which features are the most important when detecting malware using resource utilization metrics. Through this approach, we introduce the concept of dynamic malware signatures derived from process resource utilization metrics, identifying features that prominently reflect malware activity for detection purposes.

- We investigate the dynamic nature of operating systems and the role of other tenant processes as indirect indicators of malware presence, demonstrating how malware leaves evidence that assist the Transformer model in detecting infection, and compared this to the direct traces provided by the malware process itself.

The main research questions that we answer through our work are the following:

- **RQ1:** Can Transformer models detect the existence of malware effectively using process resource utilization metrics, and how do they perform compared to existing state-of-practice solutions?
- **RQ2:** How robust are Transformer models in real-world scenarios with limited data, and how does training complexity scale with dataset size compared to existing solutions?
- **RQ3:** Which process resource utilization features most prominently indicate malware presence, and how do they contribute to the decision-making process of Transformer models?
- **RQ4:** Can other tenant processes serve as indirect indicators of malware presence, with malware leaving evidence that help the Transformer model detect it by revealing signs of infection? Or is the malware process itself the sole direct indicator of its presence?

The remainder of the paper is organized as follows. Section 2 presents the related work regarding dynamic malware detection, Transformer-based malware detection, and malware detection using process resource utilization metrics. Section 3, describes the methodology of this work, the preprocessing pipeline and the dataset and input encoding used for the experiments, and presents the proposed Transformer architecture and the statistical methods used for validating the results, as well as the feature attribution methods used for explaining the model's decisions. Section 4, discusses the experimental process, the evaluation metrics and the tuning process of the Transformer and LSTM models. Section 5, presents the results of the experiments. Finally, Section 6 summarizes the findings and highlights the limitations of this work and the future work of the authors.

2. Related work

2.1. Dynamic malware detection

In the field of dynamic malware detection, researchers have explored a plethora of machine learning models and dynamic features to achieve efficient detection accuracy for both known and unknown malware variants. The authors of [24] focused on detecting metamorphic malware—a highly sophisticated type of malicious software that alters its code structure each time it replicates or infects a new system, making it particularly challenging for traditional signature-based antivirus solutions. By using API calls as the main behavioral feature, they employed an LSTM architecture for classification, achieving an overall accuracy of 95% and an F1-score of 0.83. Similarly, the work in [25] leveraged sequences of API calls and compared the performance of LSTM and Gated Recurrent Unit (GRU) architectures, achieving an accuracy of 96.8% and demonstrating the superiority of LSTMs over GRUs for this task. A more recent work utilized intrinsic features of API call sequences to enhance detection efficiency compared to traditional methods, which typically focus on API names or usage frequency [26]. Their Bidirectional LSTM model achieved a classification accuracy of 97.31% and an F1-score of 0.9724. Moving beyond API calls, the authors of [27] employed memory forensics, capturing memory dumps of suspicious processes as RGB images. Using an optimized CNN model, they achieved 98% accuracy. In a similar manner, the study in [28] converted process memory dumps into images. By utilizing a deep-stacked ensemble model and combining predictions from weak learners (CNNs) and feeding them into a Multi-Layer Perceptron (MLP) meta-learner, the study achieved 99.1% accuracy in Windows malware detection and 94.3% accuracy in Android malware detection. The research in [29] explored the use of Windows audit logs for malware detection, employing LSTMs to classify events as malicious or benign. By leveraging features like action types, process names, and target files, and comparing various embeddings such as FastText, GloVe, and Word2Vec, an accuracy of 90.84% was achieved. FastText embeddings proved particularly effective, enhancing detection by capturing semantic similarities in the data. Despite these impressive results, most of the previous approaches rely on forensic tools, reverse engineering, or specialized monitoring software to extract features. Moreover, the features used often fail to provide a holistic view of the dynamic OS environment, which, as this work demonstrates, is a crucial domain where malware behavior manifests. These methods also lack the correlative insights among dynamic features that are valuable for malware detection. Additionally, traditional machine learning techniques are constrained by their reliance on large datasets and their inability to effectively model long-term dependencies and interactions among multiple dynamic features where malware leaves distinct imprints of malicious behavior. Unlike Transformers, these methods struggle to capture the broader context and cascading effects of malware in the dynamic OS environment.

2.2. Transformer-based malware detection

Transformer-based malware detection is an increasingly popular field, with multiple researchers attempting to leverage Transformers' outstanding results from other domains (e.g. NLP) in the domain of malware detection. The authors of [30] propose a Vision Transformer architecture for malware classification, using a dataset that contains malware binaries encoded as images. They demonstrate the superiority of Transformers compared to the traditional CNN models in terms of robustness and accuracy, due to the inherent ability of Transformers to capture long-range dependencies in the input data. Their experimental results show that the proposed architecture offers several advantages over traditional CNN models, such as better performance on large-scale and complex datasets, higher interpretability, and scalability. In [31], a Transformer-based model for payload malware detection and classification, based on deep packet inspection, for detecting malicious network traffic is proposed. The authors claim that the self-attention mechanism of the Transformer allows the model to weigh significance of different parts of the sequence, providing a global understanding of data compared to other models. Their experimental results demonstrate that their approach outperforms traditional 1d-CNN, 2d-CNN and LSTM architectures. In [32], a BERT (bidirectional encoder representation from Transformers) model for detecting obfuscated malware variants is presented. Specifically, the researchers create functional obfuscated malware variants and train the Transformer with the original malware. By using adversarial training, they manage to increase the robustness and generalization of their model. Their results indicate that their approach outperforms traditional LSTM, RNN, and random forest architectures. Other research, such as [33] and [34], focuses on Android malware detection using BERT and a custom graph Transformer, respectively. Their results also indicate the superiority of Transformers compared to the so-far lead deep neural network architectures. However, most of this work focuses on static malware features for detection, while the application of Transformers in the context of dynamic malware features, such as performance metrics, remains vastly unexplored.

2.3. High-level performance metrics malware detection

The authors of [35] propose a clustering-based approach, using a modification of the sequential K-means algorithm, for detecting anomalies -malware infections- in cloud virtual machines, based on resource utilization. The main idea is to align the number of clusters with the architecture of the application being hosted on the cloud (e.g. 3-tier architecture may involve a web server, an application server and a database) and group the VMs that share similar resource utilization patterns. This way, a VM can be marked as infected, if its resource usage

is beyond a specified threshold from its cluster centroid. However, to effectively apply this approach, careful parameter tuning and expert knowledge is required. Additionally, the detection rate of this methodology is significantly lower when detecting low profile malware, which do not significantly alter resource usage but may still pose threats.

In a later work, the authors propose a new malware detection methodology using CNNs [36]. They employ resource utilization of the processes within a VM as features, instead of the VM's total resource usage, and classify them as malicious or benign. To achieve this, they train CNNs with 2- and 3-dimensional input, proving that introducing a time window as a third dimension improves the detection accuracy of the model and mitigates the mislabeling problem[1]. Nevertheless, the authors managed to achieve a detection accuracy of approximately 90%. A similar approach is followed in [37], where the researchers make use of CNNs for identifying malware using runtime utilization information, but they also employ the system's memory object information for identifying malicious processes, this way improving the detection accuracy. Additionally, they utilize the out-grafting technique[2] [38] for retrieving runtime information about the analyzed processes, this way minimizing the interaction and modifications required to be performed to the OS and the underlying hypervisor. In [22], Recurrent Neural Networks (RNNs) for detecting malware using performance metrics are employed. Compared to the aforementioned work that suggests encoding the input metrics as images, this work suggests representing a VM's behavior as a sequence of process performance metrics, including metrics such as CPU, memory, and disk utilization. It compares RNNs with CNNs, concluding that RNNs are more robust to variations in input sequences and have a higher overall performance. At the same time, it compares two different RNN models, Bidirectional (BIDI) RNNs and Long Short-Term Memory (LSTM) RNNs, concluding that while both of the models exhibit strong capabilities for malware detection, LSTM models provide a more efficient option due to their faster training times and similar performance.

Nevertheless, the literature has yet to explore the application of Transformers in the domain of malware detection using process resource utilization metrics. To the best of our knowledge, this work represents the first attempt in this area. Moreover, existing studies fail to address the application of their solutions in real-world scenarios with limited data availability, as well as the scalability of these solutions across varying volumes of data. Furthermore, prior research does not provide valuable insight into the decision-making processes of their models or the explainability of malware detection using performance metrics, leaving a knowledge gap regarding the presence of malware signatures within dynamic malware features beyond static ones. Additionally, earlier studies have not thoroughly examined the cascading effects of malware behavior within the operating system "ecosystem," a valuable information source for malware detection, as this work highlights.

3. Methodology

The experimental methodology used in this work is illustrated in Fig. 1. The process begins with data preprocessing, which includes removing unnecessary features, handling missing values, scaling, splitting the data, and encoding it into a format suitable for the employed models. For certain experiments, data sampling is required to create smaller subsets of the original dataset. Given that the dataset consists of samples derived from 104 independent experiments—each involving the execution of a distinct malware sample in a dynamic environment—sampling is performed per experiment. For instance, in the 1/10 dataset size experiment discussed in Subsection 5.2, 10% of the samples from each experiment are randomly selected. This approach ensures that the resulting dataset retains a representative distribution of samples from all 104 malware executions. In addition, it prevents overlap of samples from the same experiment across training and test folds, thereby enabling an accurate evaluation of the models' generalizability and their ability to detect zero-day malware variants.

Two model architectures are employed in the experiments: the LSTM RNN model, which has been the top-performing approach for this problem, and the proposed Transformer model. Both models are trained, and their hyperparameters are fine-tuned to optimize performance. To ensure consistency and maintain comparability with prior work, the LSTM architecture proposed in [22] is adopted with minimal modifications, where only the batch size and learning rate are fine-tuned. For the comparative analysis, the workflow—comprising data sampling, model training, and evaluation—is conducted iteratively. During each iteration, a new sample is drawn from the original dataset population, and cross-validation is performed. This iterative approach minimizes the influence of randomness on the results, ensuring robust and reliable outcomes. Additionally, the large number of iterations generates a sufficient number of results to facilitate rigorous statistical evaluation of their significance. Finally, experiments are conducted to enhance the explainability of the models using three distinct feature attribution methods. These methods provide insights into the decision-making processes of the models, further supporting their evaluation and applicability.

3.1. Data preprocessing

Data preprocessing transforms raw data into a format suitable for the employed models while improving data quality to enhance model performance. The preprocessing steps are outlined in Fig. 2. The first step includes removing features that are deemed irrelevant to the models' predictions. These include auxiliary features such as sample and experiment numbers, experiment IDs, virtual machine IDs and sample time. Sample time was excluded as there is no temporal relevance among samples from multiple experiments. Next, missing values are addressed using tailored strategies for numerical and categorical features. For numerical features, missing values are replaced with the mean value of the feature across the entire dataset, ensuring the dataset remains unbiased and the feature distributions are preserved. For categorical features, missing values are replaced with the most frequent value, reflecting the most likely category without introducing significant bias. To further enhance model performance and accuracy, both numerical and categorical features are standardized. Numerical features are normalized to have zero mean and unit variance, transforming their distribution to approximate a standard Normal distribution ($N(\mu, \sigma^2) \approx N(0, 1^2)$). Categorical features are one-hot encoded to represent discrete categories numerically without imposing ordinal relationships. These transformations reduce sensitivity to feature scaling and increase model robustness against outliers. Data are then encoded into a multivariate series format suitable for both LSTM and Transformer models. Each series is padded to ensure a standardized input size for the models. The final input shape for each series is 227×34, where 227 represents the number of processes and 34 represents the number of features for each process. Finally, the processed data are divided into training, validation, and test sets using a 60%-20%-20% split or organized into multiple folds for cross-validation, depending on the specific requirements of each experiment. This approach ensures effective model training, validation, and testing while preserving the integrity and representativeness of the dataset.

[1] The *mislabeling problem* refers to the pollution of the training data from malware samples that do not exhibit malicious behavior when analyzed, due to sophisticated malware evasion techniques used by the malware developers, e.g. the malware might detect that is being analyzed and seize its malicious behavior.

[2] *Process out-grafting* involves the relocation of a suspicious process from a production VM to a security VM, for security inspection of the process, while the system calls of the process are being forwarded to the production machine, so that its execution continues smoothly. This way, in-host and out-host security analysis problems, such as anti-malware software isolation and compatibility, are mitigated.

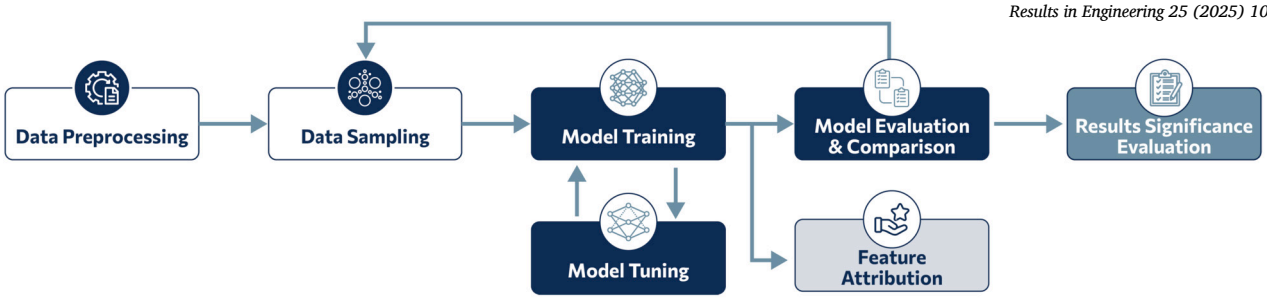

Fig. 1. Machine Learning Pipeline.

Fig. 2. Preprocessing Pipeline.

Table 1
Process Features.

Metric	Description	Metric	Description
cpu_percent	Percentage of CPU usage by the process	cpu_num	Number of CPU cores allocated to the process
cpu_sys	CPU time spent in system-space by the process	cpu_user	CPU time spent in user-space by the process
cpu_children_sys	CPU time used by child processes in system-space	cpu_children_user	CPU time used by child processes in user-space
num_threads	Number of threads the process is using	mem_shared	Amount of memory shared with other processes
mem_data	Memory used by data (excluding code and shared memory)	mem_vms	Virtual memory size used by the process
mem_rss	Memory used by the process, including code, data, and shared resources	mem_dirty	Amount of modified memory not yet written to disk
mem_swap	Amount of memory swapped to disk	mem_lib	Memory used by shared libraries
mem_uss	Unique memory used by the process, excluding shared memory	mem_text	Memory used for executable code
io_write_bytes	Total number of bytes written to disk by the process	io_read_bytes	Total number of bytes read from disk by the process
io_write_chars	Total number of characters written to disk by the process	io_read_chars	Total number of characters read from disk by the process
io_write_count	Number of write operations performed by the process	io_read_count	Number of read operations performed by the process
kb_sent	Kilobytes of data sent over the network by the process	kb_received	Kilobytes of data received over the network by the process
ionice_ioclass	I/O scheduling class for the process	ionice_value	I/O priority value for the process
nice	Priority of the process (adjusted for CPU time)	ctx_switches_voluntary	Number of voluntary context switches (process yields CPU willingly)
ctx_switches_involuntary	Number of involuntary context switches (process is preempted by the scheduler)	gid_effective	Current process group ID used for determining process's permissions
num_fds	Number of open file descriptors by the process	status_disk-sleep	Process sleeping status, while waiting for disk I/O to complete
status_running	Process running status (1 if running, 0 otherwise)	status_sleeping	Process sleeping status (1 if sleeping, 0 otherwise)

3.2. Dataset & input encoding

We apply our approach on the *Cloud Malware: VMs Performance Metrics Dataset* presented in [22], making use of performance metrics acquired in a single VM in both benign and malicious states. The selection of the dataset was made since, to our knowledge, it is the only dataset that focuses on high-level process resource utilization metrics, which can be extracted across multiple operating system environments, such as Linux-based and Windows systems, rather than being limited to low-level, hardware-specific metrics like Hardware Performance Counters (HPCs), which are out of the scope of this research. The dataset contains performance metrics for the processes of the Virtual Machine captured on various timestamps during the experiments. The process feature vector comprises performance metrics that include CPU and Memory utilization, Input/Output Read and Write, Network Traffic volume and process Status, among others. The complete feature vector for a process at a given timestamp can be found in Table 1. For each record, the dataset provides a class value, being benign or malicious.

The dataset is generated against 104 malware samples, most of them falling in the following categories: DDoS/DoS, Backdoor, Trojan, Virus, and Worm. Multiple experiments were performed each lasting 60 minutes. The first 30 minutes of each experiment are the benign phase, where random benign processes are executed within the VM. At some point within the next 10 minutes a single malware is randomly injected and the system is monitored until the 60th minute. A snapshot of the performance metrics of all processes is captured every 10 seconds, leading to a total of approximately 28000 samples, approximately 45% of them being malicious and the remaining 55% benign, resulting in a balanced dataset.

Both Transformer and RNN architectures were created for processing sequential data and capturing long-range dependencies among them. Therefore, following the approach presented in [22] we encode the input data as sequences of processes, where each process is represented by its feature vector at the given timestamp. We assume that for each timestamp $t \in T_i$ within the finite number of timestamps for each experiment $i \in [1, 104]$, we have a fixed number of processes m and a fixed size feature vector n for each process. Thus, a sample of our data is a sequence of m processes, each one represented by a feature vector of n performance metrics, as seen in Equations (1) and (2). In a similar manner to the aforestated publication, our aim is to classify each sample as benign (Equation (1)) if all the processes within the sequence are benign (bp) or as malicious (Equation (2)) if there is at least one malware

process within the sequence (*mp*) and therefore a malware is running within the system at the given timestamp. To enhance the clarity of our approach, Algorithm 1 outlines the process of extracting and encoding raw data into labeled sequences of processes, forming the final dataset samples.

$$X_t = bp_1 \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \longrightarrow bp_2 \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \cdots\cdots\rightarrow bp_m \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \tag{1}$$

$$X_t = bp_1 \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \longrightarrow mp_1 \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \cdots\cdots\rightarrow bp_m \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \tag{2}$$

```
samples = []
data = load_all_data()

for i in range(1, 104):
    experiment_data = data.get_experiment(i)

    for t in experiment_data.timestamps():
        snapshot = experiment_data.get_snapshot_at(t)
        label = 'benign'

        for process in snapshot.processes():  # m processes
            if is_malicious(process):
                label = 'malicious'
                break

        samples.append((snapshot.feature_vectors(), label)) #
      m x n
```

Algorithm 1: Encode Data

3.3. LSTMs & RNNs

Recurrent Neural Networks (RNNs) are a type of deep learning model capable of processing sequential data needed in tasks such as language translation, speech recognition, and time series prediction [39]. However, they encounter challenges like short-term memory issues, where long inputs cause the model to forget earlier information, and vanishing gradients, where gradient values diminish during back-propagation, hindering effective learning. The Long Short-Term Memory (LSTM) model is a type of RNN that was first introduced in [40] in an attempt to resolve these problems. LSTMs use a series of gates (input gate, forget gate, and output gate) to control the flow of information. This gating mechanism allows the LSTM to retain important information over long sequences and discard irrelevant data, making it particularly suitable for the above-mentioned tasks. The traditional LSTM architecture consists of a sequence of LSTM units, where each unit processes an input vector and maintains a hidden state and a cell state that gets updated at each time step. The cell state acts as a memory, and the hidden state is used for producing the output at each step. This architecture ensures that important information is propagated through the network, enabling it to learn complex temporal patterns. Although efficient, LSTMs and RNNs rely on sequential data processing, limiting their parallelization and training efficiency, while also making it challenging to capture long-range dependencies and contextual relationships in very long sequences.

3.4. Transformer models

The Transformer architecture, introduced in [21], has revolutionized the field of natural language processing and enabled significant advancements in tasks such as machine translation, text generation, and question answering. In contrast to traditional sequential models like BIDI RNNs and LSTMs, Transformers rely on a mechanism called self-attention to process input data in parallel, significantly improving computational efficiency and performance. The Transformer architecture follows an encoder-decoder structure; the encoder comprises multiple layers, each containing a multi-head self-attention mechanism and a feed-forward neural network. The self-attention mechanism allows the model to weigh the importance of different tokens in the input sequence relative to each other, capturing context more effectively. The decoder, which also comprises several layers, uses a similar structure but includes an additional attention mechanism that focuses on the encoder's output. This design enables the Transformer to handle long-range dependencies and complex relationships in the data, making it highly effective for sequence processing tasks.

3.5. Proposed transformer model

We employ a Transformer model for classifying sequential data as malicious (i.e., malware infected) or benign. For this reason, we utilize an architecture (shown in Fig. 3) that leverages the strengths of the encoder mechanism to capture long-range dependencies within the input data, while omitting the decoder mechanism, usually used in sequence-to-sequence tasks. The model comprises multiple encoder blocks, each containing multi-head self-attention mechanisms and feed-forward neural networks. Each encoder layer processes the input sequence in parallel, allowing the model to capture complex patterns and dependencies across the entire sequence efficiently. In each encoder block, the self-attention mechanism computes the relationships between different positions in the input sequence, while layer normalization and dropout are applied to improve generalization and training stability. The feed-forward part consists of convolutional layers that further transform the encoded information. This series of encoder layers ensures that the model learns rich, context-aware representations of the input data. After passing through the stacked encoder layers, the sequence representations are pooled using a global average pooling layer, which condenses the sequence information into a fixed-size vector. This vector is then fed into a multi-layer perceptron with dense layers, where further transformation and classification take place. Finally, the output layer (a softmax layer) produces the class probabilities, enabling the classification of the input sequence into one of the predefined classes. The softmax layer is proposed to additionally provide support for multi-class classification, rather than restricting the architecture to binary classification, and also to elevate the accuracy of the employed feature attribution methods, by producing a continuous output -class probabilities-, as discussed in Section 5. This architecture effectively combines the power of self-attention for capturing sequential dependencies with the efficiency and flexibility of feed-forward neural networks for classification tasks.

3.6. Paired t-test & Wilcoxon tests

For evaluating the statistical significance of the results the parametric T-test [41] and the non-parametric Wilcoxon test [42] are used. The Wilcoxon test is more robust against outliers and is better suited for smaller sample sizes where normal distribution of the data cannot be assumed, bypassing the reliance on the central limit theorem. However, for larger sample sizes (typically greater than 30) and in the absence of outliers, the T-test is more effective. The T-test utilizes parametric information, allowing it to detect smaller differences between groups with the same sample size compared to the Wilcoxon test.

3.7. Feature attribution methods

To elevate the explainability of the proposed architecture and better understand how classification decisions are made and which features have the most important role in this process, three feature attribution methods were employed: Integrated Gradients [43], Gradient x Input [44–46], and SmoothGrad [47].

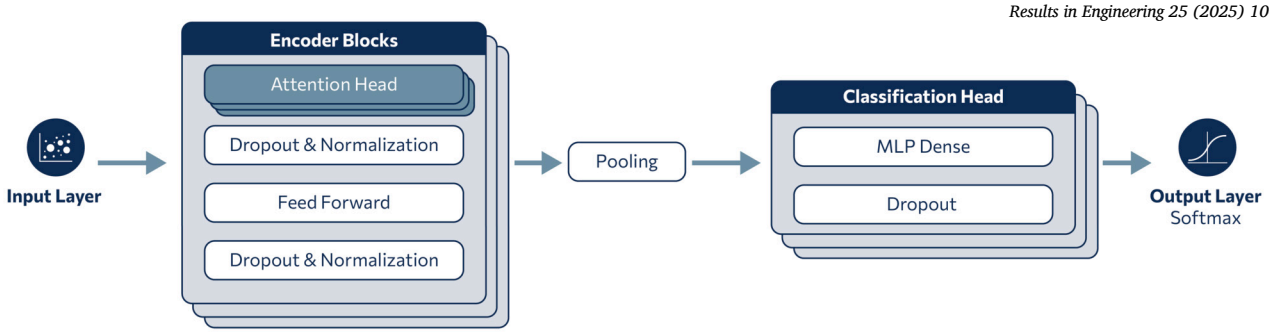

Fig. 3. Transformer Architecture for Malware Detection.

Integrated Gradients address the issue of explaining which features of the input data are most influential in the model's decision-making process. The method works by computing the gradients of the model's output with respect to the input features, but instead of using the raw input, they interpolate between a baseline (usually a neutral input like a black image or zero vector) and the actual input. Integrated Gradients then accumulate the gradients along this path to measure the contribution of each input feature to the prediction. The key idea is that this method provides a more robust attribution by averaging the changes in the model's output as the input moves from the baseline to the actual input. This ensures that the attributions are more stable and better reflect how each feature influences the prediction, making it a widely used tool for model interpretability.

The method is described as follows:

$$\mathrm{IG}_i(x) = (x_i - x_i') \times \int_{\alpha=0}^{1} \frac{\partial F(x' + \alpha \times (x - x'))}{\partial x_i} \, d\alpha \tag{3}$$

where

- $\mathrm{IG}_i(x)$ is the Integrated Gradients attribution for the i-th feature of the input x,
- x is the model's input,
- x' is the baseline input (e.g. zero vector),
- $F(x)$ is the model's prediction function,
- α is a scalar that interpolates between the baseline and the input ($a \in [0, 1]$),
- i denotes the feature index.

Gradient x Input is a straightforward feature attribution method that multiplies the gradient of the model's output with respect to the input by the input itself. This technique provides insight into how changes in input features impact the model's prediction, scaled by the feature's actual value. In doing so, it highlights the features that have both a strong influence on the output (high gradient) and a significant presence in the input (high input value). Due to its reliance on raw gradients, this method can lead to noisy attributions, as gradients are sensitive to small perturbations in the input data. This sensitivity can result in less stable attributions compared to methods like Integrated Gradients and SmoothGrad. However, because of its simplicity, Gradient x Input is computationally efficient and easy to interpret, making it an essential alternative for many real-world applications and a valuable inclusion in this study.

The method is described as follows:

$$\mathrm{GxI}_i(x) = \frac{\partial F(x)}{\partial x_i} \times x_i \tag{4}$$

where

- $\mathrm{GxI}_i(x)$ is the attribution for the i-th feature of the input x,
- x is the model's input,
- $F(x)$ is the model's prediction function,
- i denotes the feature index.

SmoothGrad enhances the stability and interpretability of gradient-based attribution methods by reducing noise. It achieves this by adding small amounts of random noise to the input and averaging the resulting attributions across multiple perturbed versions of the input. In our study, the noise is generated using a normal distribution with a mean of 0 and a standard deviation of 0.1, creating 50 noisy samples for each sample of the test data set to ensure consistent and controlled perturbations. This smoothing process helps to highlight consistent patterns in feature attributions, while suppressing distortions caused by noisy gradients.

The method is described as follows:

$$SG_i(x) = \frac{1}{N} \sum_{j=1}^{N} \frac{\partial F(x + \epsilon_j)}{\partial x_i} \tag{5}$$

where

- $\mathrm{SG}_i(x)$ is the smoothed attribution for the i-th feature of the input x,
- N is the number of noisy samples,
- x is the model's input,
- ϵ_j is a small random noise vector added to the input for the j-th sample, based on the normal distribution $N(0, \sigma^2)$,
- $F(x)$ is the model's prediction function,
- i denotes the feature index.

4. Experimental setup

For analyzing the efficiency of the proposed Transformer architecture the *Cloud Malware: VMs Performance Metrics Dataset* presented in [22] is utilized and a comparative analysis with the best-performing model on this problem, the LSTM, is performed. To the best of our knowledge, [22] is the only work on detecting malware within a single machine using process resource utilization metrics that has publicly released its dataset. For this reason, the LSTM architecture proposed in [22], is compared against our proposed Transformer architecture. The two approaches are compared both on the original dataset, as well as, on smaller samples with varying size, enabling the evaluation of their performance and robustness on diverse sample populations. To achieve a valid and conclusive result we perform each experiment multiple times, using cross validation methodologies in addition to statistical tools for comparing the significance of the results, such as Student's T-test and Wilcoxon signed-rank test. This section discusses in details the experimental setup, the models' architectures, their tuning process and the evaluation metrics.

4.1. Evaluation metrics

The target class of a sample (a sequence of processes) was decided to be 0 (Negative) for the benign samples and 1 (Positive) for the malicious ones. As such, the following metrics were defined:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{6}$$

$$Precision = \frac{TP}{TP + FP} \tag{7}$$

$$Recall = \frac{TP}{TP + FN} \tag{8}$$

$$F1\ Score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{9}$$

True Positives (TP) denote the samples that the model correctly identified as malicious and *True Negatives* (TN) are the ones that the model correctly identified as benign. On the other hand, *False Positives* (FP) denote the samples that the model falsely classified as malicious and *False Negatives* (FN) are the ones that were incorrectly classified as benign, essentially denoting how many malware infected samples the model failed to identify.

4.2. LSTM architecture & tuning process

The LSTM model presented in [22] consists of eight layers in total. These include:

- Three LSTM layers with 256, 128 and 64 units respectively, each followed by a 10% dropout layer.
- An input layer designed to accept the sequential input as described in Subsection 3.2.
- An output layer employing the softmax activation function, suitable for classification.

To optimize the LSTM model's performance, the batch size and learning rates were fine-tuned using the grid search strategy. The grid consisted of:

- Three batch size candidates: 16, 32 and 64.
- Three learning rate candidates: e^{-4}, e^{-5} and e^{-6}.

The model was built from scratch using the architecture described above, with only the batch size and learning rate hyperparameters tuned to ensure optimal performance. This resulted in a total of nine distinct models, each trained for 100 epochs to allow sufficient time for convergence. The best performing model used a batch size of 16 and a learning rate of e^{-5}, achieving a test accuracy of approximately 96.8%. By utilizing the LSTM architecture and configuration presented in [22], we achieved similar results, suggesting that the original experiment was successfully replicated.

4.3. Transformer architecture & tuning process

Transformers are generally larger and more complex models compared to LSTMs, featuring a greater number of hyperparameters that require fine-tuning. In the proposed Transformer architecture designed for classification (Fig. 3), there are eight distinct hyperparameters that need to be tuned (tested parameter range in brackets):

- *num_encoder_blocks*: [2, 4, 6]
  This parameter defines the number of encoder blocks in the Transformer. More encoder blocks can allow the model to capture more complex patterns but also increase computational complexity.
- *head_size*: [128, 256, 512]
  The size of each attention head. Larger head sizes enable each attention head to capture more detailed information from the input sequence.
- *num_heads*: [2, 4, 6]
  The number of attention heads in each encoder block. Multiple heads allow the model to focus on different parts of the input sequence simultaneously.
- *ff_dim*: [128, 256, 512]
  The number of filters in the convolutional feed-forward network within each encoder block. This parameter influences the model's capacity to learn complex representations.
- *mlp_units*: [128, 256]
  The number of units in the dense layers of the classification head. These layers additionally include dropout to prevent overfitting.
- *dropout* rate: [0.1, 0.2, 0.3]
  The dropout rate applied to the encoder blocks and the classification head. Dropout is a regularization technique to prevent overfitting by randomly setting a fraction of input units to zero during training.
- *batch_size*: [48, 64, 80, 96]
  The number of samples processed before the model's internal parameters are updated. Larger batch sizes can lead to more stable gradient estimates but require more memory.
- *learning_rate*: [10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}]
  The step size used by the optimization algorithm to update the model parameters.

Table 2
Transformers Tuning Best-2 Models.

Models	A	B
num_encoder_blocks	4	2
head_size	128	512
num_heads	6	6
ff_dim	512	256
mlp_units	[128]	[256]
dropout	0.1	0.1
batch_size	96	96
learning_rate	10^{-3}	10^{-3}
Accuracy	96.34%	96.01%
Precision	96.38%	96.05%
Recall	96.34%	96.01%
F1-score	96.34%	96.01%
Training Time	66.06 s/epoch	95.96 s/epoch

The tuning process involved selecting the optimal combination of these hyperparameters to maximize the model's performance. Each hyperparameter combination was evaluated to determine the best configuration for the classification task at hand. However, due to the large number of available models (7776), the grid search approach was rejected and the hyperband methodology was preferred for tuning. *Hyperband* is an increasingly popular hyperparameter tuning algorithm that focuses on speeding up random search through adaptive resource allocation and early-stopping, this way achieving a better time-to-convergence [48].

The hyperband tuner was left to run for 100 epochs, a sufficient choice for the models to converge. Because the best two performing models had very similar performance, we decided to perform a 10-fold cross validation with early-stopping comparing the two models regarding performance and training time. The architectures of the models as well as their averaged performance metrics are presented in Table 2.

Although the performance difference between the two models is minimal, Model A was chosen as the final architecture due to its slightly better evaluation metrics and significantly faster training time per epoch. Notably, both models converged in less than 20 epochs on average, using a 5% patience within the 100-epoch training span.

5. Experiments, results & evaluation

Three comparative and two model explainability experiments were conducted, focusing on examining the performance of the Transformer model compared to the top-performer on this problem, the LSTM model, and assessing their overall performance and robustness against variable dataset lengths. Additionally, the experiments conducted aimed at explaining the decision-making process of the models by studying how each feature impacts the model's output. At the same time, the final experiment aimed to showcase the cascading effects of malware behavior

Table 3
Transformer vs LSTM Full Dataset.

Metrics	Accuracy	Precision	Recall	F1-score	Training Time (s)
Transformer	0.9744	0.9722	0.9708	0.9715	5869.54
LSTM	0.9708	0.9696	0.9656	0.9675	6247.90
% Increase	0.37	0.27	0.54	0.41	-6.06
T-test p-value	6.63×10^{-11}	0.0019	3.61×10^{-6}	1.71×10^{-10}	0.1135
Wilcoxon p-value	1.27×10^{-9}	0.0006	1.39×10^{-5}	2.83×10^{-9}	0.0760

on system-wide processes and to explore the role of the malware process itself in detection through the proposed architecture.

The reason for performing experiments with variable sample sizes is that in the domain of malware detection using process resource utilization features, there is a lack of datasets, while augmenting existing ones with synthetic data, such as using Gaussian Noise or Bootstrapping, might not be efficient and representative, due to the diversity of malware families, generating complex signatures within these dynamic features. Therefore, experiments were also performed on smaller-sized sampled datasets, to test the robustness of the models across different dataset sizes.

The comparative experiments include performing a 10-fold cross-validation, each fold running for 200 epochs combined with early-stopping that uses a 10% patience (20 epochs), generating evaluation metrics across all folds for each model. This process is repeated multiple times (usually ten) for each experiment, resulting in a representative volume of aggregated evaluation metrics. The metrics from all folds across all experiments are then aggregated, and descriptive statistics, such as the mean, median, and standard deviation, are calculated. Finally, the two models are compared, and the statistical significance of the comparison is validated using both paired T-tests and Wilcoxon tests, to account for the small number of samples and potential outliers.

All the experiments were performed on Ubuntu 22.04, running on an 8-core AMD Ryzen 7 with 40 GB of RAM, equipped with NVIDIA GTX 1660 GPU. The GPU was utilized for training the models implemented using the keras [49] and tensorflow [50] frameworks.

5.1. *Transformer vs LSTM - original dataset size*

For the first experiment, the original dataset was utilized, preserving its size of approximately 28000 samples. The experiment included performing 10-fold cross-validation of the two models, repeated ten times, each time with a different 10-fold split. This resulted in a total of 100 samples for each model metric computed, providing a sufficiently large sample of results that can be assessed with statistical analysis tools. The averaged metrics for each model, along with the percentage increase of each metric using the LSTM model as the reference, are presented in Table 3. The table also includes the p-values of the tests measuring the significance of the differences among the metrics' means.

The results show that the Transformer models surpass the LSTM models, achieving superior performance across all metrics, highlighting the Transformers ability of capturing complex patterns within the data. However small, the difference on the performance is statistically significant, denoted by both T-test and Wilcoxon tests ($p < 0.01$). At the same time, the Transformer model outperforms the LSTM model regarding training time, decreasing the time required to train each model by approximately 6%. This is justified by the Transformers' ability of parallelization, enabling them to process entire sequences of data simultaneously, due to their self-attention mechanism, in contrast to the LSTMs that process data sequentially. The p-values indicate that there is evidence of statistical significance on the training time difference ($p \approx 0.1$)

Table 4 presents the comparison of our approach with the most relevant studies in malware detection using runtime process resource utilization metrics. This comparison clearly demonstrates that the Transformer model proposed in this work achieves the highest performance across all metrics. It not only surpasses the accuracy of existing methods but also exhibits a superior ability to identify malware samples, as evidenced by the higher recall rate. This improvement in recall is particularly crucial in real-world malware detection scenarios, where minimizing false negatives (i.e., failing to detect actual malware) is crucial. The enhanced performance of the Transformer model can be attributed to its ability to effectively capture the complex relationships within process resource utilization metrics, leading to more accurate and robust malware detection.

5.2. *Transformer vs LSTM - 1/10 dataset sample*

In the second experiment random samples are taken from the original dataset and 10-fold cross-validation is performed 10 times to assess the models' robustness and performance with smaller dataset sizes. Each experiment draws a new random sample from the original population with approximately 2800 entries, amounting to 10% of the full dataset size. The results of the experiments are presented in Table 5. Albeit the LSTM model is faster to train, the Transformer model outperforms it significantly across all metrics, achieving an increase of as low as 4.16% for Precision and up to 59.98% for Recall. The Recall metric is particularly significant in the domain of malware detection, as it measures the proportion of True Positive predictions out of all Positive samples in the dataset, highlighting a model's ability to detect malware. As a result, the LSTM's Recall of 0.5072% indicates that it can only identify half of the dataset's malware, a number unacceptable for real-life applications. In contrast, the Transformer performs significantly better, correctly identifying 81.14% of the malware. Overall, the Transformers outperform the LSTM models and prove to be significantly more robust to smaller dataset samples, suitable for applications with limited data. At the same time, while the LSTM model offers faster training times, the significant performance degradation makes it a poor choice, especially considering that the Transformer model provides vastly superior results with only a modest increase in training time.

Regarding the statistical significance of the results, although all p-values indicate the statistical importance of the differences of the averaged metrics ($p < 0.01$), this does not hold true for Precision, where the two tests notably contradict. The Wilcoxon p-value (6.85×10^{-5}) provides strong evidence against the null hypothesis, indicating that the observed difference of the models' Precision is statistically significant, while the T-test p-value indicates the opposite (0.3144). By further analyzing the data, it was observed that there is a high number of outlier values (14 out of the 100 LSTM Precision values are zero) and the data distribution is skewed, this way violating the T-test assumptions. This makes the Wilcoxon test more trustworthy, because its non-parametric nature makes it robust to outliers and skewed distributions. Therefore, by relying on the Wilcoxon test result for the Precision metric, it can be concluded that all the observed differences of the models' metrics are statistically significant.

5.3. *Transformer vs LSTM - 1/100 dataset sample*

In the final experiment, the two models are trained using a 1% random sample of the original dataset, approximately 280 samples. The experiment includes performing 10-fold cross-validation 10 times, each time drawing a new random sample from the original population. The results, presented in Table 6, highlight that limited data impacts both

Table 4
Comparison of Runtime Process Resource Utilization Metrics Methods.

Work	Model Architecture	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
[36]	CNN	≈ 90.00	≈ 90.00	≈ 87.00	≈ 92.00
[37]	CNN	96.99	94.48	93.90	94.19
[22][3]	LSTM	97.08	96.96	96.56	96.75
This Work	**Transformer**	**97.44**	**97.22**	**97.08**	**97.15**

Table 5
Transformer vs LSTM 1/10 Dataset Sample.

Metrics	Accuracy	Precision	Recall	F1-score	Training Time (s)
Transformer	0.8566	0.8671	0.8114	0.8372	556.55
LSTM	0.7617	0.8325	0.5072	0.5985	439.69
% Increase	12.46	4.16	59.98	39.88	26.58
T-test p-value	2.19×10^{-9}	0.3144	1.66×10^{-15}	3.40×10^{-12}	0.0010
Wilcoxon p-value	1.13×10^{-8}	6.85×10^{-5}	1.35×10^{-11}	4.57×10^{-11}	0.0024

Table 6
Transformer vs LSTM 1/100 Dataset Sample.

Metrics	Accuracy	Precision	Recall	F1-score	Training Time (s)
Transformer	0.7538	0.8174	0.5693	0.6466	49.48
LSTM	0.5701	0.2133	0.1713	0.1610	32.87
% Increase	32.22	283.22	232.34	301.61	50.53
T-test p-value	6.47×10^{-22}	8.85×10^{-27}	1.30×10^{-17}	2.42×10^{-26}	4.18×10^{-7}
Wilcoxon p-value	3.29×10^{-15}	6.08×10^{-15}	7.63×10^{-13}	1.23×10^{-15}	2.05×10^{-6}

models' performance, but the Transformer proves more robust, achieving a moderate Accuracy score of 75.38%, a high Precision, and a moderate Recall score. Notably, even with just 1% of the dataset, the Transformer correctly identifies 56.93% of malware, outperforming the LSTM's 50.72% Recall from the previous experiment with 10% of the dataset. Additionally, although slower to train, the Transformer delivers acceptable overall performance for applications requiring relatively small training times or when training data are limited, unlike the LSTM. The difference between the models is remarkable, from 32.22% higher accuracy, up to 301.61% higher F1-score. This establishes Transformers as far more superior models in terms of performance, when very limited data are available, but with a measurable higher training time. The validity of the aforementioned results is ensured by both T-tests and Wilcoxon tests, that denote the statistical significance of the observed differences.

5.4. Discussion

The experiments conducted provide a thorough comparison of the performance, robustness, and training times of LSTM and Transformer models in the domain of malware detection using high-level resource utilization metrics. The analysis spans three different dataset sizes—100%, 10%, and 1% of the total samples—allowing for a detailed understanding of how each model performs under varying conditions. In all experiments, the Transformer models consistently outperform the LSTMs across all measured metrics, with the sole exception of training times. This pattern becomes more pronounced as the dataset size decreases, highlighting that Transformers are more robust and less dependent on large amounts of data. On the other hand, LSTMs may struggle in scenarios where data are limited, as they fail to match the effectiveness of Transformers even when provided with larger datasets.

Transformers' superior performance can be attributed to their self-attention mechanism, which enables them to process information sequentially and weigh the importance of different parts of the input sequence. This allows them to capture long-range dependencies more effectively, which is crucial for understanding complex patterns in resource utilization data, such as intricate interactions between different system components. Furthermore, the demonstrated ability of Transformers to learn from limited data is particularly beneficial in malware detection scenarios where obtaining large, labeled datasets can be challenging. By effectively capturing global relationships from smaller datasets, Transformers can achieve competitive performance even with limited training data, making them more adaptable to real-world scenarios with data scarcity. Moreover, the ability of Transformers to capture abstract patterns and relationships within the data likely contributes to their superior generalization capabilities. This means that Transformers are better able to adapt to unseen malware variants (zero-day malware), which is crucial for effective malware detection in the ever-evolving threat landscape. Finally, the significantly higher Recall achieved by Transformers has crucial implications for real-world malware detection. Reduced False Negatives mean fewer undetected malware infections, which can significantly mitigate the impact of malware on individual users and organizations.

While Transformers exhibit superior performance, they often come at a higher computational cost compared to LSTMs, due to their increased complexity. However, this disadvantage is mitigated as the dataset size increases. The parallel processing capabilities of Transformers enable them to handle large datasets more efficiently than LSTMs, resulting in relatively faster training as data size grows. This efficiency not only accelerates the training process but also demonstrates the scalability of Transformers, making them a more suitable option for large-scale applications. The ability of Transformers to maintain high performance with smaller datasets and scale effectively with larger ones highlights their robustness and adaptability. Fig. 4 shows the training times of the two models relative to the dataset size, with both axes displayed on a logarithmic scale. The plot reveals that the increase rate in training

[3] The results depicted are the ones extracted in this work, which involved replicating the original experiment using the LSTM architecture described in [22], conducting it 100 times, and using the mean for comparison. It's important to acknowledge that the original study did not include a statistical analysis or cross-validation, potentially limiting the generalizability of their findings. Our work addresses this by employing a more rigorous methodology, including repeated cross-validation and statistical significance testing, to provide a more robust evaluation of the models' performance. Therefore, the results presented here, which demonstrate the superior performance of Transformers, are based on a more comprehensive analysis and may differ from those originally reported.

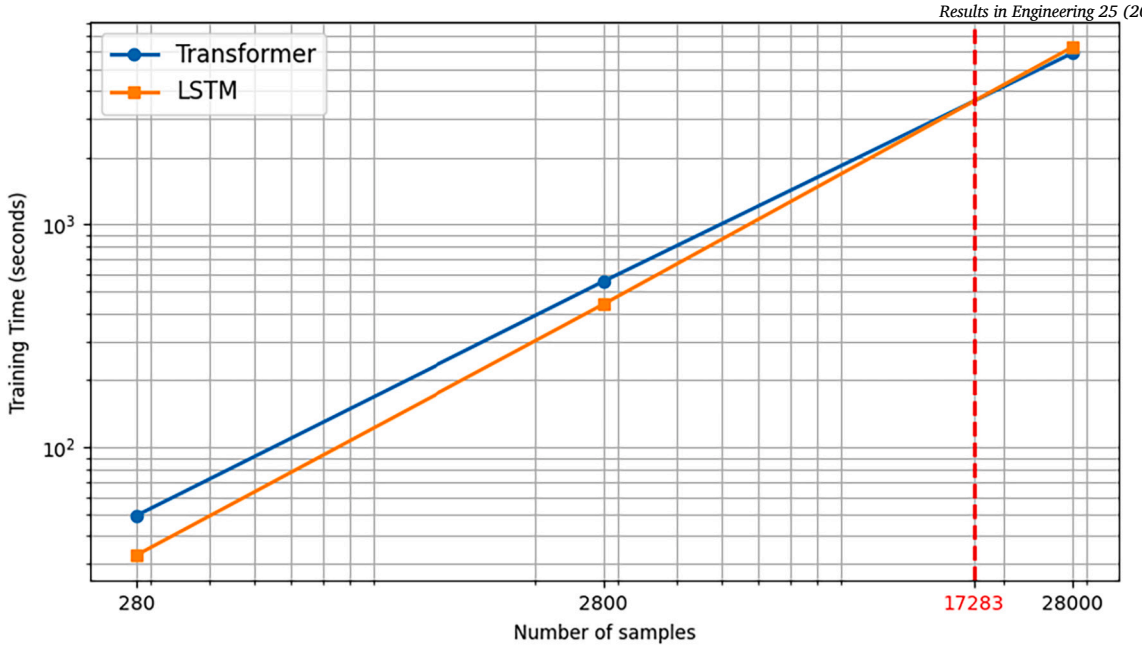

Fig. 4. Training Time vs Dataset Size.

times as the number of samples grows is higher for the LSTM model. The two lines intersect at approximately 60% of the dataset size, or around 17,000 samples, beyond which the LSTM becomes slower to train than the Transformer.

In conclusion, although LSTM models may offer quicker training times on smaller datasets, their reliance on larger datasets to achieve adequate performance, combined with their overall weaker results in terms of robustness and accuracy, limits their practicality for real-world malware detection tasks using high-level resource utilization metrics. In contrast, Transformers, despite their initial complexity and longer training times, prove to be the more robust and a reliable choice, especially in applications where data availability varies and high-level performance is crucial. At the same time, they constitute the optimal choice for larger datasets, exhibiting higher performance and smaller training times.

5.5. Transformer model feature attribution

In the previous sections, it was made clear that the Transformer models outperform the LSTMs in the scope of detection accuracy, achieving a better overall performance across all comparative experiments. However, due to their black-box nature, the features that ultimately attribute the most to the model's decision-making process remain an open question. By answering this question, we can potential hypothesize that there are specific resource utilization features, where malware creates dynamic signatures, i.e., their behavior is more expressive, which ultimately leads to their detection. To achieve that, explainable AI methodologies were employed. Specifically, three feature attribution methods were employed: Integrated Gradients, Gradient x Input and SmoothGrad, as described in Subsection 3.7, to enhance the validity of the results and demonstrate the applicability of each method within the proposed architecture. Each method has its advantages and disadvantages, including computational complexity, robustness to noise and baseline reliance, making them applicable to different real-world scenarios, depending on the specific needs of the application.

As described in Section 3, the proposed Transformer architecture generates a probability value for all target classes via the softmax function. Thus, initially two experiments were conducted using the *Integrated Gradients* method: one using the maximum softmax output (i.e., the predicted class) and another using the raw softmax output (i.e., class probabilities). Although this attribution method is designed to work with both types of outputs, we assume that the experiment using the raw softmax output is more reliable for attribution purposes. This is because the continuous nature of the raw softmax output provides a smooth gradient for Integrated Gradients. This allows the gradient to reflect subtle variations in the importance of features in all classes. In contrast, the discrete nature of the maximum class output limits the gradient's ability to capture the full spectrum of feature influences. Thus, by leveraging the softmax's continuous output, the Integrated Gradients method is likely to produce more informative and accurate feature attributions, reflecting the nuanced relationships between the input features and the model's overall confidence in its predictions. This observation is also reflected on the experiment results, where the more detailed output of the experiment that used class probabilities, enabled the model to express varying levels of importance along a broader scale (Fig. 5 and Table 7) where $IG_i \in [0, 0.4)$, compared to the non-scalar output experiment (Fig. 6) where $IG_i \in [0, 0.01)$.

Despite the differences, the results in both experiments indicate that certain resource utilization features allow malware behavior to be more expressive. Key features include process status, memory types (e.g., resident memory size (mem_rss), unique set size (mem_uss), shared memory size (mem_shared), virtual memory size (mem_vms), and text memory size (mem_text)); CPU time priority (nice), disk I/O (ionice_value), network activity (kb_received, kb_sent) and number of file descriptors (num_fds). Overall, the results suggest that malware behavior can be captured through a distinct set of features, which form the dynamic signature of the analyzed malware.

To maximize the validity of the results and extract meaningful conclusions about the most features that contribute most in malware detection, two more experiments were performed using the *Gradient x Input* and *SmoothGrad* methods. Both experiments were performed utilizing the raw softmax output (i.e., class probabilities), due to its enhanced expressiveness. The results of the two methods are shown in Fig. 7 and Fig. 8, respectively. Fig. 9 shows the aggregated importance of the features based on the three methods mentioned above using class probabilities.

The results of the experiments show that, in all of the experiments conducted using the three feature attribution methods—Integrated Gradients, Gradient × Input, and SmoothGrad—the findings converged on similar conclusions, with only minor differences in the significance ranking of individual features. The most significant attribute for classifying a sample as malicious or benign was the *sleeping status* of a process. This result highlights the fact that malware behavior is strongly expressed

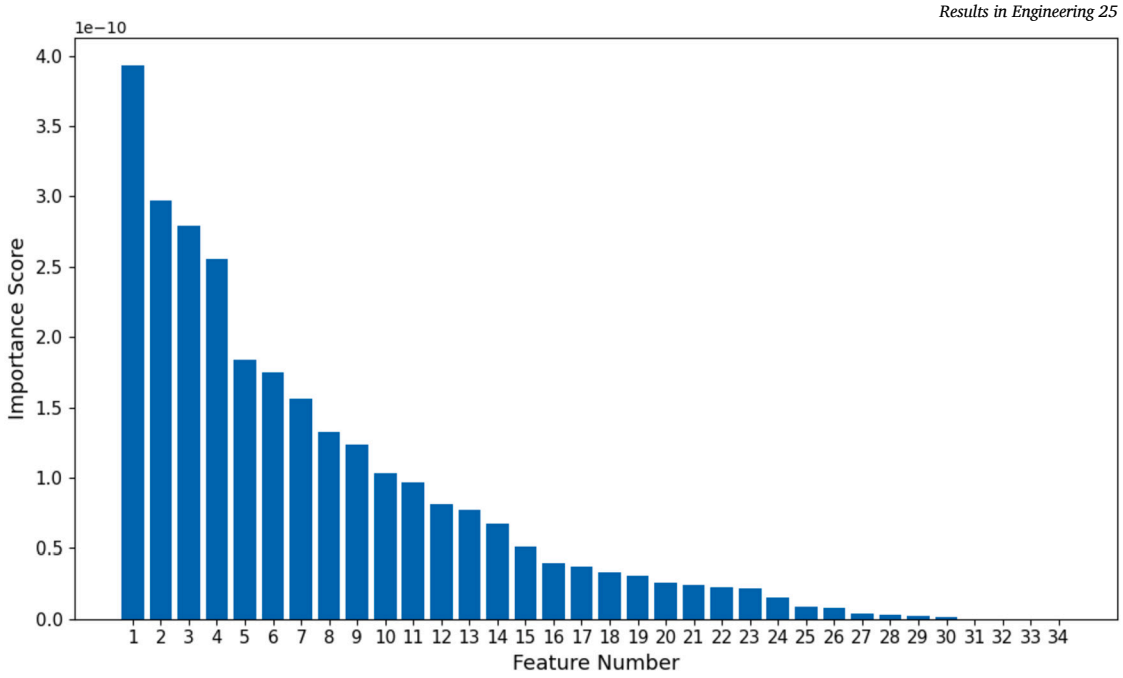

Fig. 5. Integrated Gradients Using Class Probabilities.

Table 7
Feature ordering. (For a detailed explanation of each feature's meaning, refer to Table 1.)

Number	Feature	Number	Feature	Number	Feature
1	status_sleeping	13	gid_effective	25	io_write_count
2	mem_rss	14	cpu_num	26	cpu_children_user
3	mem_uss	15	num_threads	27	io_read_bytes
4	ionice_value	16	cpu_percent	28	io_write_bytes
5	mem_shared	17	io_read_chars	29	io_write_chars
6	mem_vms	18	ctx_switches_involuntary	30	status_running
7	mem_text	19	ionice_ioclass	31	status_disk-sleep
8	nice	20	cpu_user	32	mem_dirty
9	kb_received	21	cpu_sys	33	mem_swap
10	kb_sent	22	io_read_count	34	mem_lib
11	num_fds	23	ctx_switches_voluntary	-	-
12	mem_data	24	cpu_children_sys	-	-

through the time spent by processes in a sleeping state, as well as the frequency with which they transition between active and sleeping states. Such transitions, particularly when occurring in an unusual or erratic pattern, are a distinctive characteristic of malware behavior, setting it apart from typical system processes. This conclusion aligns with the understanding that once activated, malware engages in persistent, often continuous, malicious activity, such as establishing connections with command-and-control servers, scanning the network for potential targets, or encrypting local files.

Following the sleeping status, *memory utilization* features emerged as the next most important set of attributes. In particular, the resident memory size and unique set size were the highest-ranking features across all experiments. Other notable memory-related features include shared memory, virtual memory size, and text memory used by executable code. The patterns in which processes use memory—both in the presence and absence of malware—are crucial for distinguishing malicious activity. Malware may utilize memory in patterns that are distinct from benign processes, either by consuming unusual amounts of memory or occupying memory for longer durations. These patterns can thus be a strong signal for detection. Furthermore, when a system is infected, both benign and malware processes tend to exhibit irregular memory usage behaviors (Subsection 5.6), contributing to the detection of malicious samples.

Another important finding was the role of *process priority* (specifically, nice and ionice_value) in malware detection. The presence of malware often leads to a contest for system resources, which can distort the typical patterns of resource allocation, particularly in terms of process priority. Malware, by interfering with resource distribution, can cause shifts in how processes are prioritized, which may aid in detection. Lastly, *network activity*—measured by the amount of KB sent and received—emerged as one of the top-ranking features. This is particularly logical, as many malware types, such as DDoS/DoS, Backdoors, Trojans, Viruses, and Worms, rely heavily on network communication. These malware variants use the network to communicate with command-and-control servers, propagate to other systems, or exfiltrate data. Therefore, the volume of network traffic generated serves as a strong indicator of its maliciousness.

In summary, these features collectively form a dynamic malware signature, with each feature contributing differently to the detection process. The sleeping status, memory utilization, process priority, and network activity together provide a comprehensive profile of system behavior, with each feature offering unique insights into potential malware activity. However, it is important to note that not all features are equally useful for malware detection. While memory utilization plays a critical role, other features, such as swapped memory, memory occupied by shared libraries, and modified memory not yet written to disk, showed minimal relevance. Moreover, although one might expect CPU-related features to be important—given malware's typically resource-intensive nature—our results suggest that these features do not significantly contribute to distinguishing malicious from benign behav-

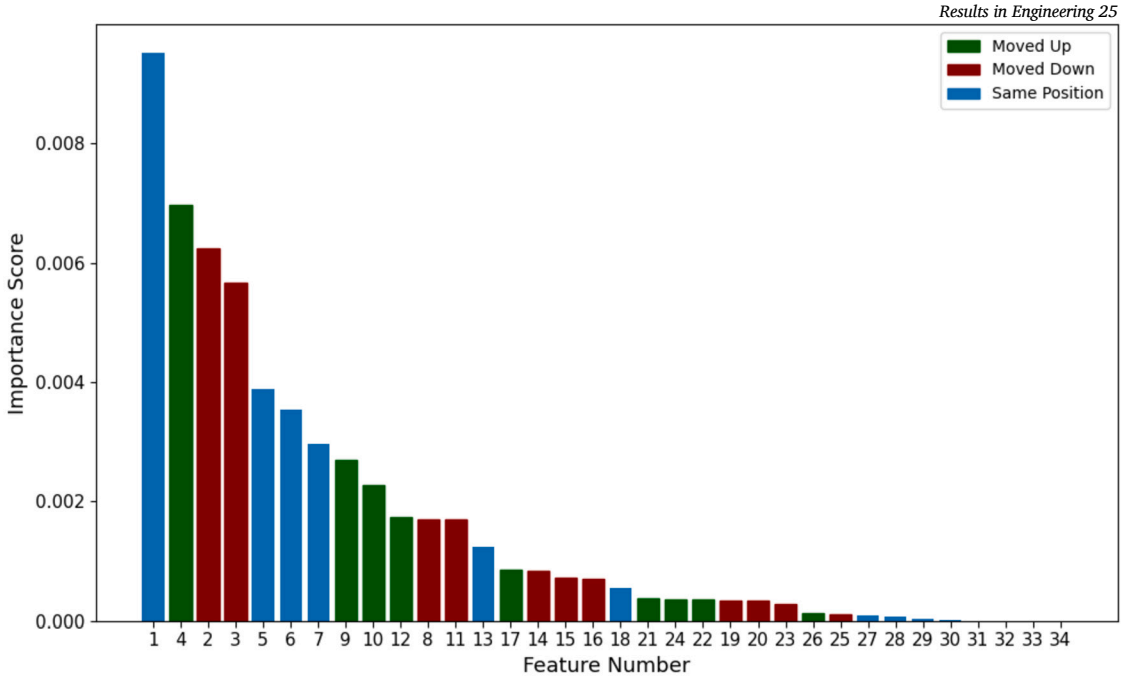

Fig. 6. Integrated Gradients Using the Predicted Class. (All comparisons are against the Integrated Gradients using Probabilities experiment (Fig. 5).)

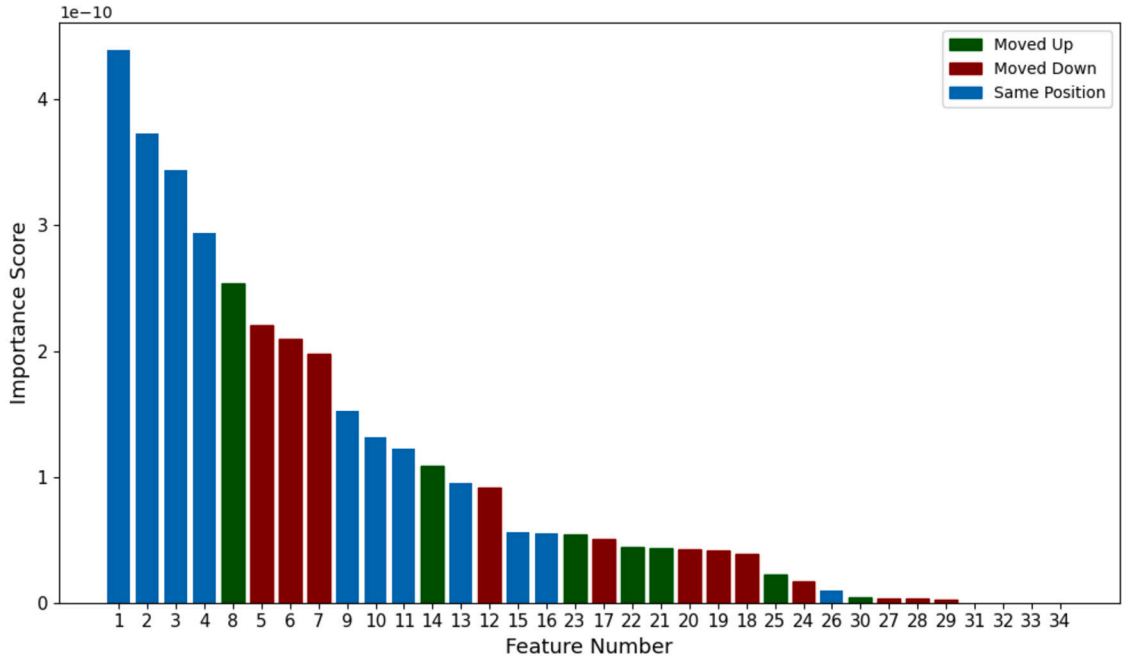

Fig. 7. Gradient x Input Feature Attributions. (All comparisons are against the Integrated Gradients using Probabilities experiment (Fig. 5).)

ior. This implies that CPU usage alone may not offer sufficient differentiation for malware detection, especially when compared to other factors like memory usage or network activity. In conclusion, this understanding enables the development of more efficient detection methods with a reduced feature space while still achieving a high level of accuracy.

5.6. Cascading effect of malware on system-wide processes

Further delving into understanding malware impact, we examine how malware affects the operating system "ecosystem" and whether malware behavior impacts other tenant processes, leaving traces on them through which malware can be detected. In other words, *does the performance of other tenant processes serve as an indirect indicator of malware presence? Or is the malware process itself the only direct signal of its presence?*

To answer this question, first we have to look at system processes that directly interact with the malware process. The most prominent candidate for this is systemd[4]. Systemd, as the process manager of the

[4] *Systemd* is an init system and service manager for Linux that is responsible for initializing the user space, mounting file systems, managing hardware, and starting and managing system services required to have a functional Linux system ([51]). Systemd is initiated directly by the kernel and owns PID 1, being the first user-mode process created. For this reason, and because all other processes are started directly by systemd or by one of its children, it is usually referred to as the "mother of all processes".

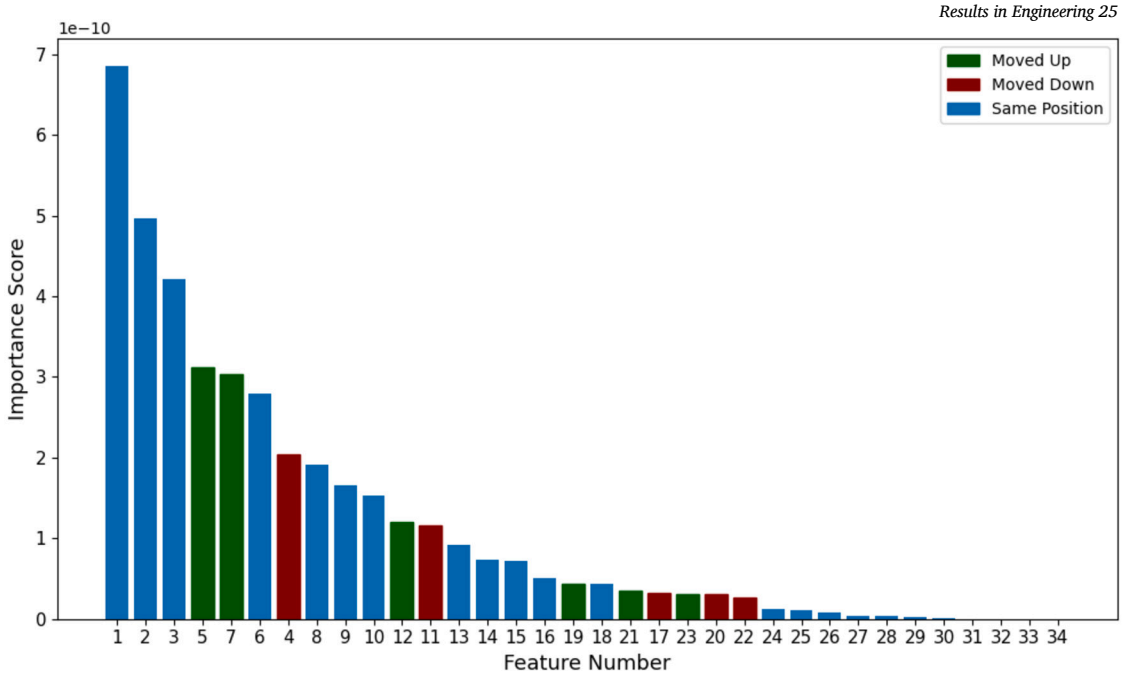

Fig. 8. SmoothGrad Feature Attributions. (All comparisons are against the Integrated Gradients using Probabilities experiment (Fig. 5).)

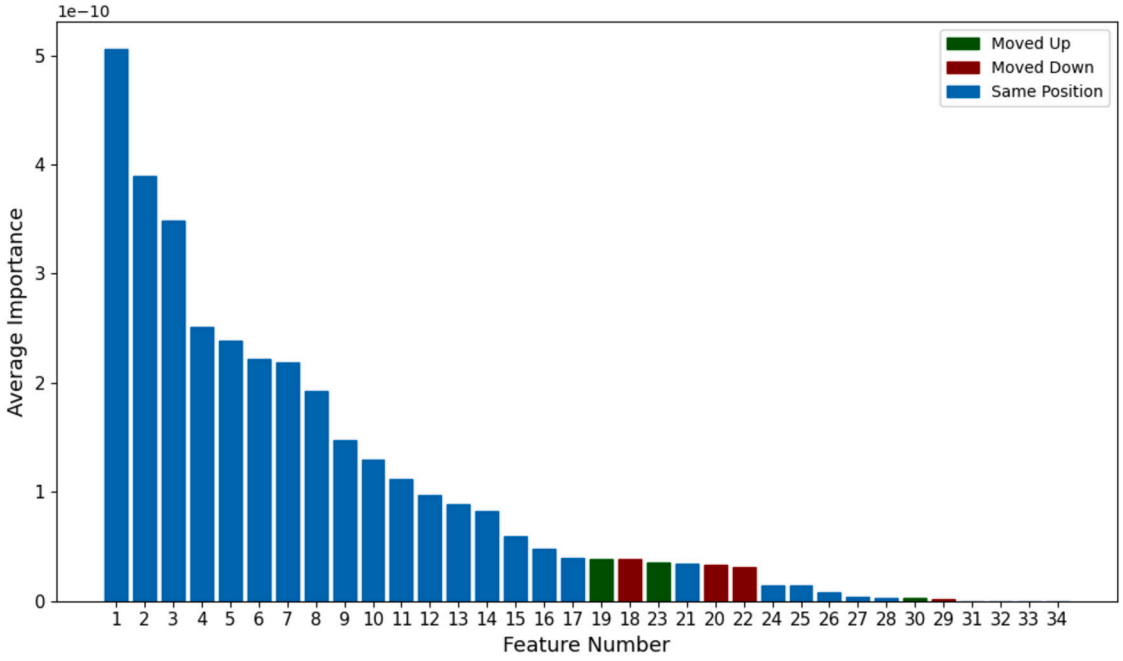

Fig. 9. Aggregated Feature Importance. (All comparisons are against the Integrated Gradients using Probabilities experiment (Fig. 5).)

Linux OS, is expected to be directly influenced by the malware process behavior. This becomes evident in Fig. 10 and Table 8, which depict memory usage of the systemd process throughout an experiment, where a Trojan[5] is inserted in the system. Memory-related metrics appear stable before the injection, suggesting that systemd was operating normally, with predicable memory consumption. Right after the malware is introduced (red dashed line), the memory consumption starts oscillating significantly. This might reflect abnormal memory requests, such as frequent memory allocations and deallocations, revealing the malware interference with the system's memory allocation mechanism. This behavior could be designed to destabilize or overload the system, in an attempt for the malware to cover its tracks or make detection harder.

As expected, malware behavior is not confined to system processes, but affects the broader "ecosystem", imprinting evidence of its behavior to other tenant processes, even when it does not directly interact with them. This can be observed in Figs. 11 (along with Table 9), 12 and 13,

[5] The inserted malware is a Trojan variant (commonly labeled as Trojan. GenericA/Cryp. Elknot), designed to infiltrate systems covertly, using encryption and obfuscation techniques to evade detection. Once active it can manipulate system resources creating fluctuations in memory, CPU, and I/O usage and also establish unauthorized remote access, allowing attackers to control the infected system, exfiltrate data, or leverage the device as part of a botnet for coordinated attacks.

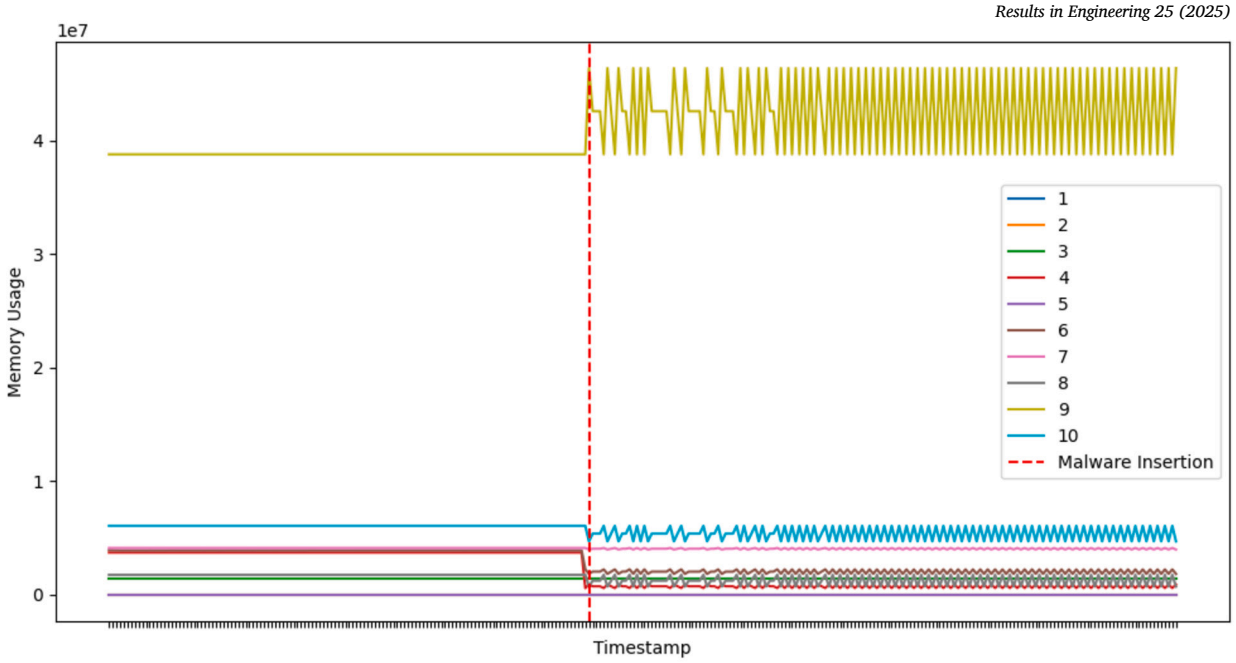

Fig. 10. Memory usage of systemd over time.

Table 8
Feature numbering of Fig. 10. (For a detailed explanation of each feature's meaning, refer to Table 1.)

Number	Feature	Number	Feature	Number	Feature
1	mem_swap	5	mem_dirty	9	mem_vms
2	mem_lib	6	mem_pss	10	mem_rss
3	mem_text	7	mem_shared	-	-
4	mem_uss	8	mem_data	-	-

Table 9
Feature numbering of Fig. 11. (For a detailed explanation of each feature's meaning, refer to Table 1.)

Number	Feature	Number	Feature	Number	Feature
1	mem_swap	4	mem_uss	7	mem_shared
2	mem_lib	5	mem_dirty	8	mem_data
3	mem_text	6	mem_pss	9	mem_rss

depicting memory usage, CPU percentage and ionice[6] value of a daemon process used to handle PHP script execution (php-fpm7.0). Similar to systemd, vibrant oscillations are observed in memory-related metrics, suggesting unstable memory allocation behavior. This is because malware may trigger frequent memory reallocations or increase system-wide memory pressure, forcing php-fpm7.0 to adjust its memory usage to compete for resources. CPU usage exhibits a much greater degree of fluctuation following the malware insertion. Specifically, frequent drops occur, as the process is periodically deprived of CPU resources. These irregular patterns suggest that the malware might be indirectly causing php-fpm7.0 to experience higher computational demands or interruptions in its execution flow. Additionally, the malware seems to impact the I/O scheduling of the system, affecting processes' access to disk I/O resources. The drops in the ionice value after malware injection indicate that php-fpm7.0 is being granted higher I/O priority to cope with increased contention or instability. This suggests that the malware's influence on disk I/O is severe enough to prompt the system to prioritize certain processes dynamically. This behavior highlights malware's broad impact on resource allocation and system performance.

To validate our hypothesis and explore whether these unique patterns play a pivotal role in detecting malware, we conducted the following experiment. First, the top-performing Transformer architecture, as presented in Sections 3 and 4, was trained on 60% of the dataset samples, with its performance evaluated across the entire dataset, including both seen and unseen samples. Then, a modified dataset was generated from the original, where all malware processes within each malicious sample were removed[7]. Finally, the previously trained model was evaluated on this newly generated dataset. In both of the experiments, the models produced similar results, as shown in the confusion matrix (Fig. 14), where 1 and 0 respectively denote the presence and absence of malware within a sequence of processes (i.e., a sample). This consistency indicates that the Transformer model's accuracy remained unaffected by the removal of malware processes. This was due to the malware's behavior being fully manifested within the other processes at an extent that the malware process itself was not necessary anymore for detection of the malware's presence. This holds true even for processes that lack direct interaction with the malware process, revealing infection patterns that Transformers can effectively capture for accurate malware detection.

It should be noted that in contrast to the aforementioned experiment -chosen to highlight cases with pronounced changes in other tenant processes behavior- there are cases where malware behaviors are less apparent (i.e., patterns are not as prominent). This variation arises because malware behavior differs among variants and even more across families. For example DDoS/DoS malware often requires substantial CPU, bandwidth and memory to generate enough traffic to cripple a target. One the other hand, backdoors are typically designed to avoid detection and consume minimal resources to remain hidden. Another cause of this variation is the evasion technique used by the malware, with some resource-intensive types attempting to hide their presence by causing

[6] The ionice value is a setting in Linux systems that determines the I/O scheduling priority of a process, meaning it controls how much priority a process has for input/output operations (such as reading and writing to disk). The ionice value for each process ranges from 0 to 7, where a smaller value indicates a higher priority.

[7] In order to maintain the dimensionality of the formulated problem and avoid biasing the dataset, each malware process feature was replaced by the mean value, which is zero due to the standardization performed on the data, aligning with the methodology followed for padding.

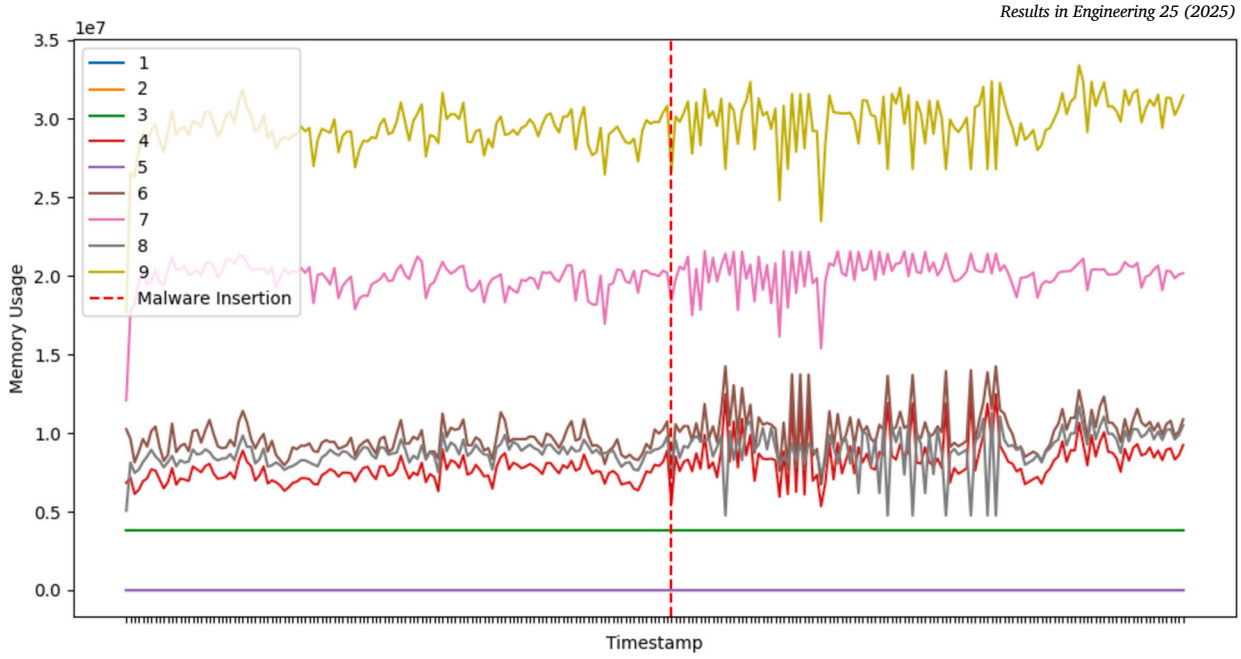

Fig. 11. Memory usage of php-fpm7.0 over time.

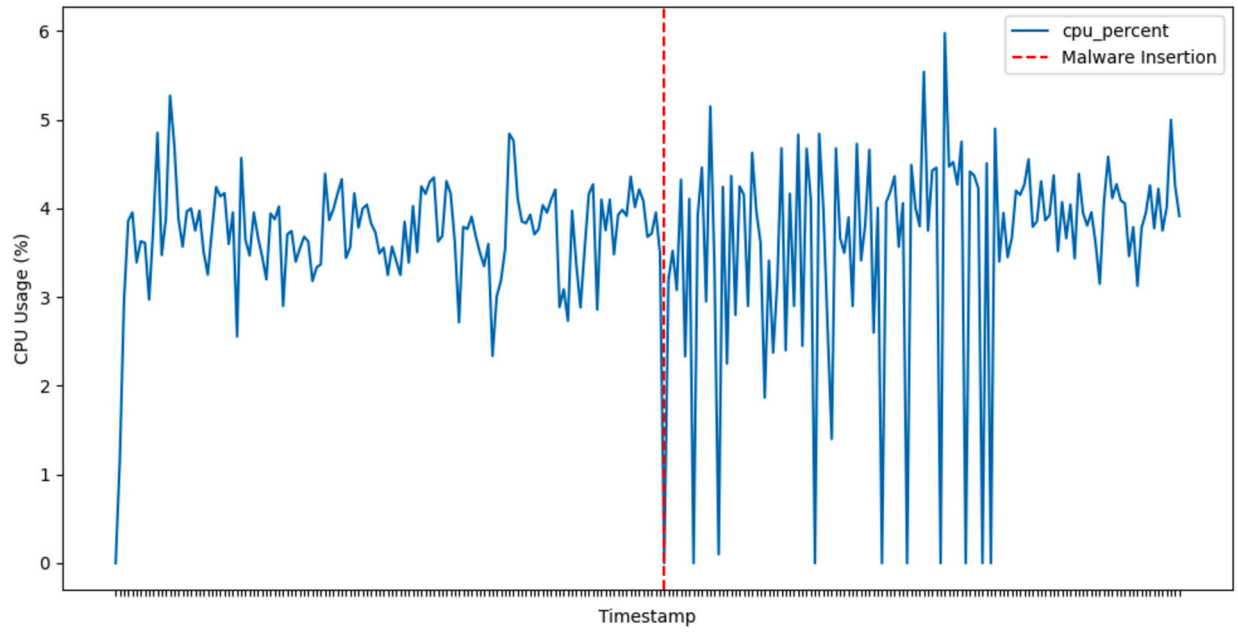

Fig. 12. Cpu usage of php-fpm7.0 over time.

major system disruptions, while others maintain a low profile to avoid detection. However, this does not seem to affect the proposed Transformer architecture, which successfully captured even the more nuanced patterns and achieved state-of-the-art detection accuracy. Overall, we conclude that tenant processes within the same OS act as primary indicators of malware presence, in terms of performance metrics for malware detection.

In conclusion, the above results demonstrate strong potential for detecting malware infections by analyzing the behavior of other tenant processes. This approach enables the detection of malware even when sophisticated evasion techniques are used, such as malware altering its name, spawning new processes, or attaching to existing ones. This effectiveness arises from the operating system functioning as an "ecosystem," where each process behavior influences others due to the limited resources they share. This observation suggests that malware detection can benefit from a broader perspective, examining indirect signs of infection across the system rather than focusing solely on direct indicators of malware presence. Such a strategy not only strengthens detection, but also enables detection models to adapt to increasingly sophisticated evasion techniques, reinforcing their robustness in real-world scenarios.

6. Conclusions, limitations & future work

In this work, we presented a Transformer-based architecture for malware detection using process resource utilization metrics. Utilizing the Hyperband tuning algorithm, we optimized the hyperparameters and conducted a comparative analysis with the leading LSTM model, using rigorous statistical analysis methodologies. Furthermore, we examined the robustness of the two models, and how their performance is affected by variable dataset sizes. Additionally, we explored the decision-making

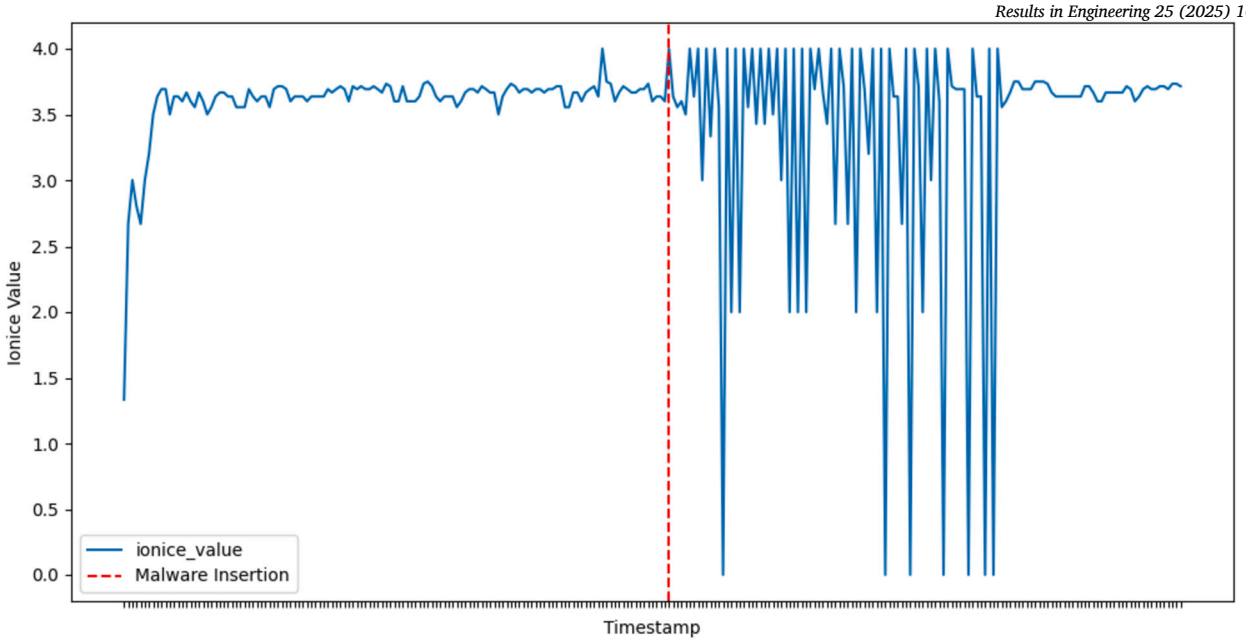

Fig. 13. Ionice value of php-fpm7.0 over time.

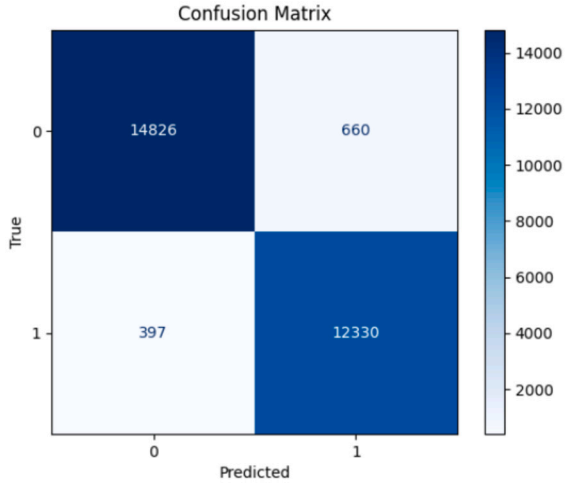

Fig. 14. Transformer evaluation on dataset with and without malware processes.

process of the Transformer architecture, using three feature attribution methods. Finally, we examined whether other tenant processes within the operating system may serve as indirect indicators of malware presence. Our results indicate that Transformers demonstrate strong capability for detecting the existence of malware using process resource utilization metrics, outperforming the leading model on the problem, the LSTM (***RQ1***). Furthermore, the proposed architecture managed to maintain high performance with smaller data samples, perform better under conditions of limited data availability, and scale effectively with larger datasets, making it the most suitable option for real-world applications (***RQ2***). Moreover, the feature attribution experiment revealed that there exists a specific set of resource utilization features where malware behavior is manifested (***RQ3***), indicating the existence of dynamic malware signatures within those features, similar to the static signatures usually found in malware's binaries. Ultimately, we highlighted the cascading effect of malware behavior to other system tenant processes, which acts as the primary indicator of malware presence, enabling for efficient detection using the proposed architecture (***RQ4***). Thus, we argue that the aforementioned results establish Transformers as the new state-of-the-art model for the discussed research problem.

Our findings have significant implications for the field of cybersecurity. The demonstrated superiority of Transformers over traditional ML approaches in handling real-world scenarios with limited data availability and scaling effectively with larger datasets paves the way for developing highly adaptable, efficient, and robust malware detection systems capable of addressing evolving cyber threats. This advancement enables the creation of solutions tailored for environments ranging from resource-constrained IoT devices to large-scale cloud infrastructures, ensuring efficient protection against sophisticated malware attacks. The concept of dynamic malware signatures in process resource utilization, introduced in this work, along with the identification of key attributing features, opens up numerous possibilities. By identifying the most prominent indicators of malicious activity—such as process status, memory usage, CPU and disk priority and network usage—security frameworks can be streamlined to focus on the most expressive and actionable features. This insight facilitates the design of lightweight detection models for specific environments like IoT devices, where computational resources are limited, while simultaneously enhancing the generalizability of detection mechanisms across diverse platforms. The cascading effects of malware on tenant processes are particularly significant for advancing malware detection, especially against sophisticated evasion techniques such as name altering, process injection, or low-profile behavior. By leveraging the indirect disruptions malware causes across system processes, detection mechanisms can remain effective even when direct indicators are obscured. These findings not only strengthen the robustness of dynamic malware detection, but also lay the foundation for resilient cybersecurity solutions that adapt to evolving threats, ensuring efficient protection in increasingly complex cyber-physical environments.

However, two limitations were identified in this study. The first is the lack of multiple datasets for further testing, which would strengthen the validity of the results. This limitation arises because the domain of malware detection using high-level process resource utilization metrics (such as CPU, memory, and disk usage) is relatively new. The first work addressing this problem and releasing a related dataset was [22], which provided the dataset used in this study. While the authors of [37] claimed to have created their own dataset using a similar data extraction process, to our knowledge, it has not been made publicly available. Secondly, while this study approached the problem of malware detection holistically, the complex and diverse nature of different malware families necessitates a more focused examination. This targeted approach has the potential to enhance detection accuracy and leverage explain-

able AI methods, ultimately providing deeper insights into how each malware family behaves and manifests within resource utilization features.

Our future work includes addressing the problem of identifying the malware family of the infection. Additionally, we plan to examine the behavior of distinct malware families to identify unique characteristics that may make them more susceptible to detection. This will enable us to further address the question of whether malware leaves detectable dynamic signatures on systems' resources, similar to the static signatures found in malware's binaries and commonly used in malware detection, and which resources are the most expressive and efficient for each malware family. Finally, we aim to generate our own datasets to include a broader range of malware families beyond those in the current dataset.

The code we used to evaluate the proposed Transformer-based architecture can be found in https://github.com/dcnatsos/transformer-based-malware-detection.

CRediT authorship contribution statement

Dimosthenis Natsos: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Conceptualization. **Andreas L. Symeonidis:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

Funding: This work was supported by the European Commission through the work programmes Research Analysis Identifier SystEm (RAISE project No 101058479) and Nostradamus (project No 101134888).

Appendix A. List of abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
BERT	Bidirectional Encoder Representation from Transformers
BIDI	Bidirectional
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DDoS	Distributed Denial of Service
DL	Deep Learning
DNN	Deep Neural Network
DoS	Denial of Service
FP	False Positives
FN	False Negatives
GloVe	Global Vectors for Word Representation
GRU	Gated Recurrent Unit
HPC	Hardware Performance Counter
I/O	Input/Output
ID	Identifier
IoT	Internet of Things
LSTM	Long Short-Term Memory
ML	Machine Learning
MLP	Multi-Layer Perceptron
NLP	Natural Language Processing
OS	Operating System
PID	Process ID
RNN	Recurrent Neural Network
RGB	Red Green Blue
RQ	Research Question
TN	True Negatives
TP	True Positives
VM	Virtual Machine
Word2Vec	Word to Vector
XAI	Explainable AI

Data availability

We have shared a link to our code within the document

References

[1] W.A. Jansen, T. Winograd, K. Scarfone, Guidelines on Active Content and Mobile Code, NIST Special Publication 800-28 Revision 2 National Institute of Standards and Technology, 2008, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-28ver2.pdf.

[2] A. Morchid, Z. Oughannou, R.E. Alami, H. Qjidaa, M.O. Jamil, H.M. Khalid, Integrated Internet of things (iot) solutions for early fire detection in smart agriculture, Results Eng. 24 (2024) 103392, https://doi.org/10.1016/j.rineng.2024.103392, https://www.sciencedirect.com/science/article/pii/S259012302401644X.

[3] K. M., S. M., R. Banakar, Evolution of iot in smart vehicles: an overview, in: 2015 International Conference on Green Computing and Internet of Things (ICGCIoT), 2015, pp. 804–809.

[4] Y.J. Qu, X.G. Ming, Z.W. Liu, X.Y. Zhang, Z.T. Hou, Smart manufacturing systems: state of the art and future trends, Int. J. Adv. Manuf. Technol. 103 (2019) 3751–3768, https://doi.org/10.1007/s00170-019-03754-7.

[5] B.N. Silva, M. Khan, K. Han, Towards sustainable smart cities: a review of trends, architectures, components, and open challenges in smart cities, Sustain. Cities Soc. 38 (2018) 697–713, https://doi.org/10.1016/j.scs.2018.01.053, https://www.sciencedirect.com/science/article/pii/S2210670717311125.

[6] M. Campoverde-Molina, S. Luján-Mora, Cybersecurity in smart agriculture: a systematic literature review, Comput. Secur. 150 (2025) 104284, https://doi.org/10.1016/j.cose.2024.104284, https://www.sciencedirect.com/science/article/pii/S016740482400590X.

[7] S. Sharma, B. Kaushik, A survey on Internet of vehicles: applications, security issues & solutions, Veh. Commun. 20 (2019) 100182, https://doi.org/10.1016/j.vehcom.2019.100182, https://www.sciencedirect.com/science/article/pii/S2214209619302293.

[8] N. Tuptuk, S. Hailes, Security of smart manufacturing systems, J. Manuf. Syst. 47 (2018) 93–106, https://doi.org/10.1016/j.jmsy.2018.04.007, https://www.sciencedirect.com/science/article/pii/S0278612518300463.

[9] F.A. Aboaoja, A. Zainal, F.A. Ghaleb, B.A.S. Al-rimy, T.A.E. Eisa, A.A.H. Elnour, Malware detection issues, challenges, and future directions: a survey, Appl. Sci. 12 (2022), https://doi.org/10.3390/app12178482, https://www.mdpi.com/2076-3417/12/17/8482.

[10] J. Singh, J. Singh, Challenge of malware analysis: malware obfuscation techniques, Int. J. Inf. Secur. Sci. 7 (2018) 100–110.

[11] O. Or-Meir, N. Nissim, Y. Elovici, L. Rokach, Dynamic malware analysis in the modern era—a state of the art survey, ACM Comput. Surv. 52 (2019), https://doi.org/10.1145/3329786.

[12] R. Lyda, J. Hamrock, Using entropy analysis to find encrypted and packed malware, IEEE Secur. Priv. 5 (2007) 40–45, https://doi.org/10.1109/MSP.2007.48.

[13] I. Santos, F. Brezo, B. Sanz, C. Laorden, P.G. Bringas, Using opcode sequences in single-class learning to detect unknown malware, IET Inf. Secur. 5 (2011) 220–227, https://doi.org/10.1049/iet-ifs.2010.0180.

[14] J.Z. Kolter, M.A. Maloof, Learning to detect and classify malicious executables in the wild, J. Mach. Learn. Res. 7 (2006), https://doi.org/10.1145/1014052.1014105.

[15] T.-Y. Wang, C.-H. Wu, C.-C. Hsieh, Detecting unknown malicious executables using portable executable headers, in: 2009 Fifth International Joint Conference on INC, IMS and IDC, 2009, pp. 278–284.

[16] Z.-G. Chen, H.-S. Kang, S.-N. Yin, S.-R. Kim, Automatic ransomware detection and analysis based on dynamic api calls flow graph, in: Proceedings of the International Conference on Research in Adaptive and Convergent Systems, RACS'17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 196–201.

[17] S. Wang, Z. Chen, Q. Yan, B. Yang, L. Peng, Z. Jia, A mobile malware detection method using behavior features in network traffic, J. Netw. Comput. Appl. 133 (2019) 15–25, https://doi.org/10.1016/j.jnca.2018.12.014, https://www.sciencedirect.com/science/article/pii/S1084804518304028.

[18] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, S. Stolfo, On the feasibility of online malware detection with performance counters, in: Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA'13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 559–570.

[19] Z. Xu, S. Ray, P. Subramanyan, S. Malik, Malware detection using machine learning based analysis of virtual memory access patterns, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, 2017, pp. 169–174.

[20] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, H. Liu, A review of Android malware detection approaches based on machine learning, IEEE Access 8 (2020) 124579–124607, https://doi.org/10.1109/ACCESS.2020.3006143.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, https://arxiv.org/abs/1706.03762, 2017, https://doi.org/10.48550/arXiv.1706.03762, arXiv:1706.03762.

[22] J.C. Kimmel, A.D. Mcdole, M. Abdelsalam, M. Gupta, R. Sandhu, Recurrent neural networks based online behavioural malware detection techniques for cloud infrastructure, IEEE Access 9 (2021) 68066–68080, https://doi.org/10.1109/ACCESS.2021.3077498.

[23] S. Karita, N. Chen, T. Hayashi, T. Hori, H. Inaguma, Z. Jiang, M. Someki, N.E.Y. Soplin, R. Yamamoto, X. Wang, S. Watanabe, T. Yoshimura, W. Zhang, A comparative study on transformer vs rnn in speech applications, in: 2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU), 2019, pp. 449–456.

[24] F.O. Catak, A.F. Yazı, O. Elezaj, J. Ahmed, Deep learning based sequential model for malware analysis using windows exe api calls, PeerJ Comput. Sci. 6 (2020) e285, https://doi.org/10.7717/peerj-cs.285.

[25] W.R. Aditya Girinoto, R.B. Hadiprakoso, A. Waluyo, Deep learning for malware classification platform using windows api call sequence, in: 2021 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS, 2021, pp. 25–29.

[26] C. Li, Q. Lv, N. Li, Y. Wang, D. Sun, Y. Qiao, A novel deep framework for dynamic malware detection based on api sequence intrinsic features, Comput. Secur. 116 (2022) 102686, https://doi.org/10.1016/j.cose.2022.102686, https://www.sciencedirect.com/science/article/pii/S0167404822000840.

[27] M.R. Naeem, M. Khan, A.M. Abdullah, F. Noor, M.I. Khan, M.A. Khan, I. Ullah, S. Room, A malware detection scheme via smart memory forensics for windows devices, Mob. Inf. Syst. 2022 (2022) 9156514, https://doi.org/10.1155/2022/9156514.

[28] H. Naeem, S. Dong, O.J. Falana, F. Ullah, Development of a deep stacked ensemble with process based volatile memory forensics for platform independent malware detection and classification, Expert Syst. Appl. 223 (2023) 119952, https://doi.org/10.1016/j.eswa.2023.119952, https://www.sciencedirect.com/science/article/pii/S0957417423004542.

[29] M. Ring, D. Schlör, S. Wunderlich, D. Landes, A. Hotho, Malware detection on windows audit logs using lstms, Comput. Secur. 109 (2021) 102389, https://doi.org/10.1016/j.cose.2021.102389, https://www.sciencedirect.com/science/article/pii/S0167404821002133.

[30] I. Ben abdel ouahab, L. Elaachak, M. Bouhorma, Enhancing malware classification with vision transformers: a comparative study with traditional cnn models, in: Proceedings of the 6th International Conference on Networking, Intelligent Systems & Security, NISS'23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1–5.

[31] K. Stein, A. Mahyari, G. Francia, E. El-Sheikh, A transformer-based framework for payload malware detection and classification, in: 2024 IEEE World AI IoT Congress (AIIoT), 2024, pp. 105–111.

[32] F. Lu, Z. Cai, Z. Lin, Y. Bao, M. Tang, Research on the construction of malware variant datasets and their detection method, Appl. Sci. 12 (2022) 7546, https://doi.org/10.3390/app12157546.

[33] F. Ullah, A. Alsirhani, M.M. Alshahrani, A. Alomari, H. Naeem, S.A. Shah, Explainable malware detection system using transformers-based transfer learning and multi-model visual representation, Sensors 22 (2022) 6766, https://doi.org/10.3390/s22186766.

[34] Y. Fan, M. Ju, S. Hou, Y. Ye, W. Wan, K. Wang, Y. Mei, Q. Xiong, Heterogeneous temporal graph transformer: an intelligent system for evolving Android malware detection, in: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining, KDD'21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 2831–2839.

[35] M. Abdelsalam, R. Krishnan, R. Sandhu, Clustering-based iaas cloud monitoring, in: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), 2017, pp. 672–679.

[36] M. Abdelsalam, R. Krishnan, Y. Huang, R. Sandhu, Malware detection in cloud infrastructures using convolutional neural networks, in: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), 2018, pp. 162–169.

[37] D. Tian, R. Zhao, R. Ma, X. Jia, Q. Shen, C. Hu, W. Liu, Mdcd: a malware detection approach in cloud using deep learning, Trans. Emerg. Telecommun. Technol. 33 (2022) e4584, https://doi.org/10.1002/ett.4584.

[38] D. Srinivasan, Z. Wang, X. Jiang, D. Xu, Process out-grafting: an efficient" out-of-vm" approach for fine-grained process execution monitoring, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, 2011, pp. 363–374.

[39] L.C. Jain, L.R. Medsker, Recurrent Neural Networks: Design and Applications, 1st ed., CRC Press, Inc., USA, 1999.

[40] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (1997) 1735–1780, https://doi.org/10.1162/neco.1997.9.8.1735.

[41] Student, The probable error of a mean, Biometrika (1908) 1–25, https://doi.org/10.2307/2331554.

[42] F. Wilcoxon, Individual Comparisons by Ranking Methods, Springer, New York, New York, NY, 1992, pp. 196–202.

[43] M. Sundararajan, A. Taly, Q. Yan, Axiomatic attribution for deep networks, in: D. Precup, Y.W. Teh (Eds.), Proceedings of the 34th International Conference on Machine Learning, in: Proceedings of Machine Learning Research, vol. 70, PMLR, 2017, pp. 3319–3328, https://proceedings.mlr.press/v70/sundararajan17a.html.

[44] Captum, Input x gradient, https://captum.ai/api/input_x_gradient.html, 2019. (Accessed 3 January 2025).

[45] A. Shrikumar, P. Greenside, A. Shcherbina, A. Kundaje, Not just a black box: learning important features through propagating activation differences, arXiv preprint, arXiv:1605.01713, 2016, https://doi.org/10.48550/arXiv.1605.01713.

[46] K. Simonyan, A. Vedaldi, A. Zisserman, Deep inside convolutional networks: visualising image classification models and saliency maps, arXiv preprint, arXiv:1312.6034, 2013, https://doi.org/10.48550/arXiv.1312.6034.

[47] D. Smilkov, N. Thorat, B. Kim, F. Viégas, M. Wattenberg, Smoothgrad: removing noise by adding noise, arXiv preprint, arXiv:1706.03825, 2017, https://doi.org/10.48550/arXiv.1706.03825.

[48] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, A. Talwalkar, Hyperband: a novel bandit-based approach to hyperparameter optimization, J. Mach. Learn. Res. 18 (2018) 1–52, http://jmlr.org/papers/v18/16-558.html.

[49] F. Chollet, et al., Keras, https://keras.io, 2015.

[50] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: large-scale machine learning on heterogeneous systems, https://www.tensorflow.org/, 2015, https://doi.org/10.48550/arXiv.1603.04467, software available from tensorflow.org.

[51] D. Both systemd, Apress, Berkeley, CA, 2020, pp. 379–410.