EAGLEEYE: Attention to Unveil Malicious Event Sequences from Provenance Graphs

Philipp Gysel*, Candid Wüest*, Kenneth Nwafor*†, Otakar Jašek*, Andrey Ustyuzhanin‡*, Dinil Mon Divakaran§
*Acronis Research, †Constructor Technology,
‡Constructor University, Bremen, §Institute for Infocomm Research (I²R), A*STAR

***Abstract*—Securing endpoints is challenging due to the evolving nature of threats and attacks. With endpoint logging systems becoming mature, provenance-graph representations enable the creation of sophisticated behavior rules. However, adapting to the pace of emerging attacks is not scalable with rules. This led to the development of ML models capable of learning from endpoint logs. However, there are still open challenges: i) malicious patterns of malware are spread across long sequences of events, and ii) ML classification results are not interpretable.**

To address these issues, we develop and present EAGLEEYE, a novel system that i) uses rich features from provenance graphs for behavior event representation, including command-line embeddings, ii) extracts long sequences of events and learns event embeddings, and iii) trains a lightweight Transformer model to classify behavior sequences as malicious or not. We evaluate and compare EAGLEEYE against state-of-the-art baselines on two datasets, namely a new real-world dataset from a corporate environment, and the public DARPA dataset. On the DARPA dataset, at a false-positive rate of 1%, EAGLEEYE detects ≈89% of all malicious behavior, outperforming two state-of-the-art solutions by an absolute margin of 38.5%. Furthermore, we show that the Transformer's attention mechanism can be leveraged to highlight the most suspicious events in a long sequence, thereby providing interpretation of malware alerts.

***Index Terms*—Malware, Provenance graph, Transformer**

I. INTRODUCTION

The rate of cyber attacks, their sophistication, the number of targets, and the losses the attacks cause, have clearly witnessed a worrying upward trend over time. In May 2024 alone, more than 9 million new malware samples were reported [1]. Reports estimate that cybercrime will cost the world more than 9 trillion USD in 2024 [2]. With a mature underground market, cybercriminals have easy access to new, better, and advanced malware to assist them with their nefarious goals.

Detection of malware infection is challenging. With wide adoption of TLS, much of the network communication is encrypted today, thus limiting the capabilities of perimeter defense systems such as firewalls, deep packet inspection, and intrusion detection systems [3], [4], [5], [6], [7]. This limitation is turning the point of observation onto endpoints, where we have the capability to monitor and log low-level kernel calls of all activities, including process creations, file system activities, and network connections. EDR (endpoint detection and response) solutions such as CrowdStrike Falcon [8], SentinelOne Singularity [9], and Trend Micro Apex One [10] can log such behavior. These logs present a rich source of information useful for security auditing [11]. Hence, despite the computational and storage costs involved, security vendors have been developing and enhancing EDR solutions that provide high visibility into activities happening on endpoints via continuous logging. Importantly, EDR solutions are complemented with signatures to detect known malware, which are updated continuously as new malware samples are discovered.

Yet, simple rules that, say, match against hashes of known malicious binaries and IP addresses with bad reputation, are not sufficient to counter emerging threats. Sophisticated attacks exploit or disguise as benign applications, encrypt individual binary segments, employ polymorphic techniques, obfuscate network communication to exfiltrate sensitive information, and use other evasion techniques. This observation led to the development of *provenance graphs* as a dependency structure depicting entities (such as processes, files, etc.) at endpoints and their relationships. With such graphs, a security analyst can visually see the relationship between system entities and identify suspicious patterns [12]. These graphs also facilitate the writing of more sophisticated signatures which can be used to detect malware behavior spread across a provenance graph (see Section II for an illustration).

While signatures crafted by security analysts are precise in detecting known malware, there are important disadvantages. i) Analyzing graphs and manually writing rules is not a scalable approach, given that there are more than 300,000 new malware samples detected daily [1]. ii) Moreover, the provenance graphs are themselves huge in size [13]. iii) Attacks keep evolving with time, and with attackers now getting powerful AI tools to assist them in the process (e.g., FraudGPT [14]), the quantity and quality of attacks are only expected to rise, thus compounding the challenge. Consequently, researchers have been developing machine learning models for investigating incidents, triaging, and detecting malware [15], [16], [17], [18], [19], [20], [21], [22], [23], [24].

The following challenges are yet to be addressed. The malicious behavior of a malware sample can be spread out in a (potentially) long sequence of events in a provenance graph; existing works use models that are not able to effectively learn from such long sequences of events. This results in low detection rates. Second, it is not only important to classify behavior as malicious or not, but also to provide

Corresponding author: Philipp Gysel (gyselph@gmail.com). This work was done when all authors were affiliated with Acronis Research.

2159-1245/24/$31.00 ©2024 IEEE
DOI 10.1109/eCrime66200.2024.00009

interpretation of results, so that an analyst can efficiently assess the criticality of an incident. Addressing these challenges, we develop EAGLEEYE, a novel system that i) uses rich features from behavior events of applications, including command-line strings, ii) extracts and embeds long sequences of events, and iii) employs a lightweight Transformer model that learns to predict the intent of an application. We summarize our contributions:

- We develop a novel system called EAGLEEYE, which processes low-level events and learns the behavior of applications. We explore the capability of Transformers in modeling endpoint behavior based on process provenance graphs (Section IV).
- We enrich a provenance graph with features capturing the context of behavior events (Section V-C). Different from previous works, EAGLEEYE employs another Transformer to generate embeddings of command-line strings used to start new processes, thus capturing command details and hierarchy in the process tree (Section V-D).
- We perform a systematic evaluation and compare EAGLEEYE with state-of-the-art solutions, namely, DeepCASE [22] and ProvDetector [23]. Our experiments on two datasets demonstrate that, in comparison to the baselines, EAGLEEYE achieves significantly higher detection accuracy at very low false positive rates.
- We evaluate the modeling capability of the Transformer model in EAGLEEYE and compare it with other ML models (Section VI-C3), such as LSTMs used in existing works [15], [17], [25]. We also study the impact of command-line embeddings and security features in achieving high performance (Section VI-C4).

Moreover, we carry out a case study, which reveals that EAGLEEYE can both learn the malicious context from a long sequence and provide an explanation for the identified malicious pattern (Section VII). To support further research, we publish the code of EAGLEEYE, along with the new malware dataset we generated[1].

II. IMPORTANCE OF FEATURE-RICH PROVENANCE GRAPHS

Provenance graphs are a powerful tool to differentiate between malicious and benign applications. In this section, we demonstrate the importance of encoding detailed behavior information into the graph nodes. For this purpose, we proceed to analyze a concrete malware type and its behavior, which provides insights into how a malware detection system can identify the intent of an application. During this process, we show how a professional security analyst would craft a behavior signature for this specific malware.

Figure 1 shows the most relevant parts of the provenance graph of the well-known Raccoon Infostealer [26]. We downloaded a sample instance from VirusTotal [27], executed it in a controlled environment, and tracked its runtime behavior using a commercially available EDR tool. In what follows, we describe its behavior according to the MITRE ATT&CK

[1]Implementation and malware dataset: https://github.com/gyselph/eagle-eye

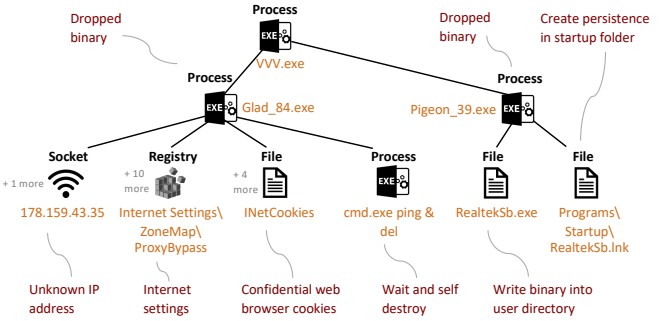


Fig. 1: Provenance graph of Raccoon Infostealer malware

framework [28]. After initial access of executable `VVV.exe`, an application `Glad_84.exe` is written to disk and started. The application first reaches out to a C&C server. In order to detect if it is within a sandbox environment, the application checks various registry keys. As part of the collection phase, the application reads multiple web browser cookies, which potentially contain confidential information. Finally, to evade detection, it executes `cmd.exe` with appropriate flags to delete the binary. To ensure that this indicator removal happens *after* the executable has completed its work, a prolonged `ping` operation is prepended to the delete step. In parallel, a second application `Pigeon_39.exe` is also written to disk and started. To gain persistence, a link file is written to the startup directory. This link file points to a dropped binary `RealtekSb.exe`, which is written to the user directory for masquerading purposes.

Security vendors typically create behavior signatures manually. For the studied malware sample, such a rule could be summarized as follows: i) The provenance graph contains at least one dropped binary that is executed. ii) A connection is established to a new and potentially suspicious IP address. iii) Multiple registry keys related to host settings are read. iv) Multiple sensitive files are read. v) Persistence is created via a dropped binary in the startup directory. vi) The main file deletes itself through a ping sleep & delete command.

As one can observe, this behavior signature uses both global information as well as fine-grained details of individual actions. For example, condition 1 above requires global knowledge spread across multiple nodes of the provenance graph, to check if a started executable was previously persisted in the same graph. On the other hand, condition 3 can only be checked if the exact registry key path is known. Finally, conditions 4 and 5 above mandate that the precise file path for each file event is given.

Next, we switch our focus to a benign provenance graph from a `Chrome.exe` instance. Figure 2 shows its key behavior, which we captured on an endpoint under real-world conditions, using the same EDR tool as for the malware sample. As shown in the figure, the web browser starts two independent Zoom sessions. The first session performs a short query to a Zoom data center, checks the local language dictionary, uses a prefetch file for faster execution, and starts a nested Zoom session. The second session is only used for

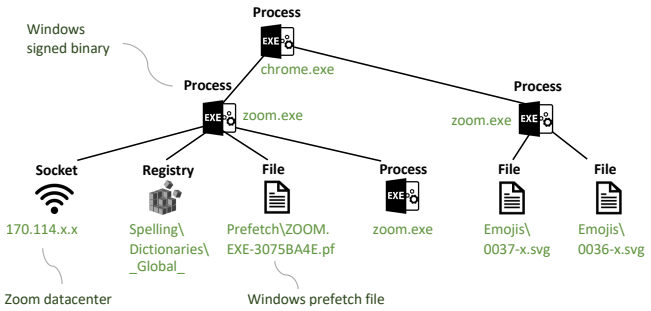


Fig. 2: Provenance graph of a Chrome instance

chatting, and emojis are used during typing.

With detailed information present in each behavior node, both the benign and malicious provenance graphs can be easily categorized by a security expert. Unfortunately, most existing solutions drop the rich information from the graph nodes and only keep the entity types (e.g., `file`, `process`, `socket`, etc.) and associated actions (e.g., `read`, `write`, `execute`, etc). As a case in point, let us consider the final file creation of both provenance graphs. With the help of all event details, the last file creation of the Raccoon Infostealer is potentially suspicious, as a link file is created in the Autostart folder, pointing to a dropped binary. However, if this event was only represented by the entity type (`file`) and the associated action (`write`), this event would look identical for both provenance graphs, thus opening the door for evasion by attackers [29].

In fact, most state-of-the-art proposals for malware detection use coarse behavior features. ProGrapher [24] and Unicorn [20] consider only the entity and action type. As already mentioned, this approach causes the last file event of both aforementioned provenance graphs to look identical. Similarly, DeepCASE [22] uses high-level SIEM events, where each event is represented by one categorical feature. By contrast, in our work, we propose to use a large feature vector per behavior event, of size 60, where the feature vector contains rich information typically used by experienced security analysts (see Section V-C). We argue that this additional and relevant information makes it possible to clearly differentiate graphs like those in Figures 1 and 2 into malicious and benign ones.

We conclude that tracking low-level kernel calls of an application, and keeping detailed features about each such behavior event, allow a human analyst to classify applications into benign and malicious. Therefore, we provide the same behavior data to EAGLEEYE, with the goal of detecting malicious behavior on endpoints.

III. THREAT MODEL

Our threat model is similar to previous works which use endpoint log analysis for malware detection [11], [23], [21], [30], [22], [24]. We assume a system in place that monitors kernel-level calls, such as registry events, network connections, file creations, etc. Commercially available EDR tools are capable of logging such events for security purposes.

We consider attacks on endpoint computers achieved by malware. We refer to the provenance graph corresponding to a malware's action on an endpoint as a malicious graph. Thus, the goal of our work is to perform online detection of malware on endpoints. A malware sample might be of any family or variant, and as such might have different goals, such as infecting other machines, or exfiltrating sensitive data. During this process, the malware sample may also be involved in other activities. In particular, many malware variants are known to disguise their intention by instrumenting legitimate processes to achieve their goals. This so-called Living-off-the-land technique [31], [32] can be used to inject code into an existing legitimate system process, or to start a new legitimate process which performs part of the malicious work. Despite this obfuscation, the kernel-level driver sees the API requests performed by the legitimate process and logs such information. Similar to past works, we assume that the auditing framework and log files are not compromised, via the use of a trusted computing base and hardening techniques.

IV. EAGLEEYE: ARCHITECTURE AND MODEL

Figure 3 presents an overview of EAGLEEYE. The left side of the figure illustrates the feature extraction mechanism and the process of transforming system logs into a data format suitable for ML modeling. As shown on the right side of the figure, the enriched and preprocessed data is then fed into the trained model for malware classification. Next, we present the modeling aspects of EAGLEEYE.

A. Transformer for learning behavior events

Since the seminal work by Vaswani et al. [33], Transformers have demonstrated their capability in various natural language processing (NLP) tasks. The Transformer architecture, which is based on the self-attention mechanisms, has shown significant efficacy, surpassing the previous state-of-the-art models such as recurrent neural networks (RNNs), making it the de facto standard in NLP today. However, past works on malware detection use RNNs [16], [22], [25], [17], [15]; we argue that Transformer models exhibit a significantly better capability for classifying computer applications, as also confirmed by our experiments (Section VI-C3).

Problem of long sequences: Malware detection is akin to the problem of finding a needle in a haystack. In the example of the Raccoon Infostealer from Figure 1, only a handful of events reveal its malicious intent. However, over the lifespan of the Raccoon executable, more than 1,000 behavior events are observed. Therefore, a significant part of the behavior sequence is irrelevant for malware detection. Transformers can deal much better with long sequences than RNNs, thanks to their attention mechanism [33].

Problem of complex context: Application classification requires detection of complex patterns, which in turn depends on multiple correlated events that are spread far apart in a sequence. Revisiting the Raccoon Infostealer (Figure 1), it is the *combination* of C&C communication, reading of sensitive files, persistence creation, and final uploading of data that turns the behavior into a malicious pattern. If 1-2 behavior events are missing, the pattern may be categorized differently

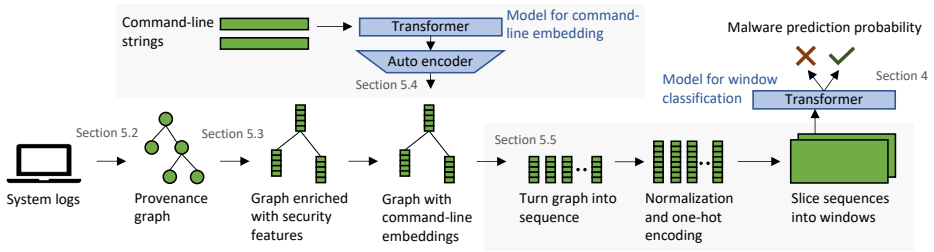


Fig. 3: System architecture of EAGLEEYE

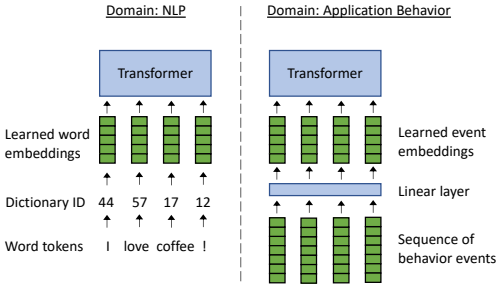


Fig. 4: Token embedding: NLP tasks vs. EAGLEEYE

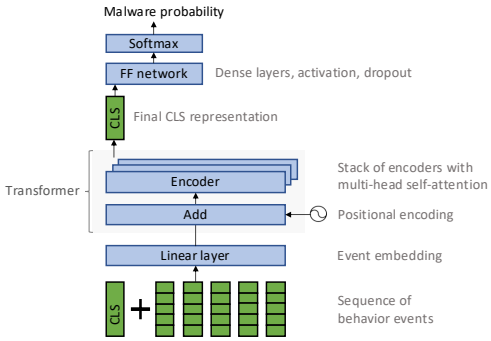


Fig. 5: Transformer architecture of EAGLEEYE

and as benign. Transformers are able to extract such complex patterns, as the internal attention mechanism attends to the whole input sequence for context creation. As we show in Section VII, EAGLEEYE's model indeed pays high attention to suspicious events while ignoring unimportant ones.

B. Embedding behavior events

EAGLEEYE represents a behavior event as a fixed-size vector of security features. Figure 4 illustrates our event embedding approach (right), and compares it with the traditional word embedding in the NLP domain (left). In the NLP setting, an embedding layer translates token IDs into word embeddings, which are learned during the training phase. In our setting, each behavior event is represented as a fixed-size vector. Our architecture contains a linear embedding layer, which learns (via back-propagation) to project the behavior events into a new latent feature representation. With this embedding layer, we reduce the dimension of each behavior vector from ∼200 to an embedding vector of dimension 60, effectively reducing the sparse high-dimensional space to a smaller latent space. Next, we describe the architecture of the proposed Transformer model.

C. Transformer architecture

There exist three Transformer architecture types: *encoder-only*, encoder *and* decoder, and *decoder-only* architecture. The first type is suited for classification problems, whereas the second and third are more complex architectures used for generative tasks. Since malware detection is a classification task, we choose to use an encoder-only architecture. Our goal is to build a lightweight malware classification system, which mandates a small-sized resource-efficient model. Therefore, we build a small Transformer model, which is based on BERT-Tiny [34].

We present the overall architecture of EAGLEEYE's Transformer model in Figure 5. The model input consists of an event sequence, where each event is represented by security features (as described in the next section). Similar to the original BERT model, we prepend each sequence with a `CLS` token. The event sequence plus `CLS` token are passed through a linear projection layer for event embedding. The positional encoding in EAGLEEYE is based on the event sequence index. The resulting sequence goes through an encoder stack with multi-head self-attention. In the final encoder layer, only the `CLS` token's representation is used, and all other tokens are discarded. The `CLS` token representation is passed through a 2-layer dense network, followed by a softmax layer, which produces the final malware classification probability.

V. EAGLEEYE: BEHAVIOR MODELING

A. Overview of data processing pipeline

Figure 6 illustrates the transformation of security logs into graph-based sequences, which serve as input to EAGLEEYE.

1) First, we deploy a system that tracks all low-level kernel calls of running applications on endpoints. To avoid an overload of data, we apply a filtering mechanism, and only collect a subset of all incoming event types (Section V-B).

2) We convert security logs into provenance graphs, which represent behavior events via raw features (Section V-B).

3.1) We enrich a provenance graph with security relevant features (Section V-C). During this process, we go beyond raw features and extract advanced features that learn context from other parts of the provenance graph.

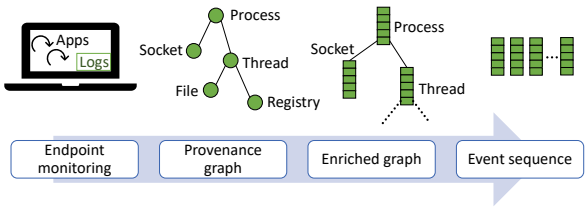


Fig. 6: Application behavior modeling

TABLE I: Behavior event types collected by EDR solutions

Processes/threads	Miscellaneous	Suspicious
Process start	Socket	Stack pivot
Process termination	File access	Memory protection
Thread creation	File permission	Hook injection
Thread suspension	Registry key	Process hollowing
Thread resumption	RPC call	Token manipulation

3.2) A novel aspect of EAGLEEYE is the inclusion of command-line embeddings, which represent commands and their hierarchy in the process tree (Section V-D).

4) Finally, a graph gets converted to a sequence of chronologically ordered nodes (Section V-E). As each node is represented by a fixed-size vector of security relevant features, the application behavior is now modeled via a sequence.

B. Convert a system log to provenance graphs

Under real-world conditions, user endpoints host a variety of applications, where each application typically produces hundreds of behavior events every minute. Similar to previous works [17], [23], [30], [24], we use provenance graphs as an intermediate data representation. EAGLEEYE builds provenance graphs by creating a node for each behavior event and edges for capturing the parent-child relationships between the different events. We represent provenance graphs as follows: we store all behavior information in nodes, and edges do not contain any information. Nodes contain information about both the action taken (e.g. read, write, execute) as well as the involved system entity (e.g. file name, path, file access mode, etc). A provenance graph's head node represents the bootstrapping process. All its child nodes are actions taken directly by this process, including the creation of new processes. Child processes, in turn, will point to new nodes containing actions initiated by them. Clearly, the provenance graph thus created is a DAG (directed acyclic graph).

EDR solutions gather various details for each behavior event, beyond just process names and call hierarchies. We use only a subset of all available information, which we store as *raw* features in nodes of the corresponding graph. Such raw features can be simple in nature, like file names and process names, which are also used in other works [23]. We also store other particulars like file access flags and command-line flags.

Every operating system has its own bootstrapping processes, which are involved during startup, and later used for orchestration of compute loads. On Windows OS, user processes are started directly or indirectly by a handful of bootstrapping processes, like `winlogon.exe`, `wininit.exe`, and `logonui.exe`. Since different applications like `notepad.exe` and `outlook.exe` should be analyzed separately, we do not include the bootstrapping processes as a common ancestor, and instead create disjoint provenance graphs for the user applications.

To construct provenance graphs from a system log, we select a random behavior event, and iteratively move up the parent-child hierarchy until we reach a native OS process. This corresponding child node (with a native OS process as parent) serves as the head of the provenance graph. The head node will only have outgoing edges and no incoming ones. We now perform breadth-first search from the graph head. For each process, we add all child processes as child nodes to the graph. Moreover, for each process, we search for all initiated kernel calls, like file accesses and network connections, and add these events as child nodes to the corresponding process node. The resulting provenance graph contains a subset of all behavior events from the system log, as there are multiple disjoint applications running concurrently on a given system.

Next, we pick another behavior event which did not get added to the previous set of provenance graphs, and proceed to create another provenance graph in the same manner. We continue building provenance graphs until each behavior event is either represented in one of the graphs, or was initiated from the operating system and is thus ignored. Thanks to this grouping of data into different graphs, we achieve separation of data, which can then be analyzed independently. Each graph gives an account of where the application interactions originate from. In the example of the Raccoon Infostealer (Figure 1), the provenance graph explains where the malicious persistence `RealtekSb.lnk` comes from, and thus the head process `VVV.exe` from the graph can be flagged as suspicious.

Our data processing pipeline has several built-in measures to control the size of the resulting graphs. First, we do not include processes from the operating system. Next, we do not include all existing behavior event types, but only the most relevant ones from a security perspective. Table I gives a list of behavior events captured by most EDR systems; however we limit EAGLEEYE to only use four event types (see Table VII in the Appendix). Finally, we set a maximum duration for each provenance graph, and start a new graph after the timer expires. While a longer duration helps to capture more dependencies and relationships among the different events, the resulting larger graph also creates computational and storage challenges. Furthermore, the goal of EAGLEEYE is to detect malicious software as early as possible, before any harm is inflicted on the endpoint.

C. Graph enrichment with security features

There are two main goals during the graph enrichment phase. i) Add more context-aware information to each graph node, which will provide EAGLEEYE with relevant information for malware detection. ii) The *raw* features of graph nodes

should be turned into preprocessed *security* features, which are in a structured format suitable for machine learning.

During the enrichment phase, EAGLEEYE leverages the context of the whole graph to condense global information into features of individual nodes. As an example, the process call hierarchy may reveal important details of an application's intent [31], [11]. We therefore use the process hierarchy and each process' command and flags to create call-chain aware command-line embeddings (Section V-D). As another example, consider *dropped binaries*, i.e., an executable *recently* downloaded from the Internet, written to disk, and then started as a new process. We define a boolean feature for process nodes to indicate whether an executable was dropped.

Each behavior event type has its own set of *raw* features. These raw features are turned into structured *security* features. A full list of our 24 security features is included in Appendix A, Table VII. As a case in point, for each registry access, we propose a security feature which indicates whether a particular registry key is creating persistence for an application. To achieve this, we leverage the full registry key path. This feature simplifies the task for the subsequent learning model, when compared to using the raw registry key path. As an important observation, there are other registry keys with the opposite effect, such as notify keys. Notify keys create a graphical notification for the user, and are typically used by benign applications; we represent notifications using a boolean feature.

As another case in point, for representation of file access events, we introduce a security feature which captures whether the file in question contains sensitive information, such as credit card data, passwords, or web cookies. In addition, we use a categorical security feature that represents the location of a given file. For this purpose, we split the whole file system into coarse groups which share a common semantic meaning. For Windows, one such important path is the autostart directory. Other important path categories include temporary directories, system directories, and user directories.

The graph enrichment step is a novel component of EAGLEEYE, which, to the best of our knowledge, is missing in existing detection solutions. Previous works either do not consider a detailed provenance graph [25], [15], [22] or use a provenance graph which contains only *raw* features [17], [23], [20], [24], [21], [35], [30].

D. Command-line embedding

Process creations, and in particular the command-line strings used for process starts, are relevant for malware detection. In fact, some recent works [31], [36] rely *only* on command-lines to perform malware detection. To demonstrate the importance of command-lines, we investigate the wait-and-self-destroy operation of the Raccoon Infostealer mentioned earlier. The full command-line string with the executable path, the name, and all flags is shown in Figure 7. A security analyst will consider this command as suspicious, since a benign program would rarely hide a wait operation in a `ping` command and subsequently delete itself. The challenge here

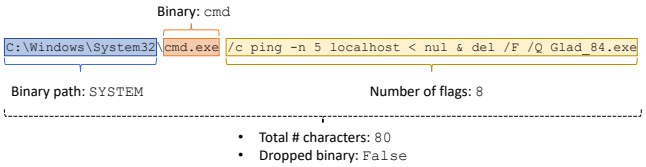


Fig. 7: Featurized representation of a command-line string

is to represent this data in a format that is understandable by a model. Typically such command-line strings are represented by handcrafted features, like the string length, the path, and the name of the binary, etc. However, such features may not capture all relevant information.

Another approach is to represent the whole string using a certain embedding. Filar et al. [36] used the word frequency algorithm TF-IDF for this purpose, and in [31] word2vec embedding [37] was applied. However, the Transformer architecture has demonstrated a superior capability for text reasoning, with models like BERT [38] being used to generate embeddings for security tasks such as phishing email detection [39].

For our representation of commands, EAGLEEYE uses a tiny `all_MiniLM_L6_v2` Transformer [40], which is pre-trained on a large corpus of data to generate text embeddings. As input to this sentence Transformer, we use the whole command-line string of both the current and parent processes. Thus the model captures: i) richer semantics than what handcrafted features do, ii) the process hierarchy. As output of the Transformer, we compute the mean of all output tokens, resulting in a 384-dimensional vector. To compress this vector into a more manageable size, we train an autoencoder in unsupervised fashion. That is, we pass all command-line strings from the training dataset through the sentence Transformer, and forward the output to the autoencoder, which then learns to compress the command-line representation into a 16-dimensional vector. The resulting vector serves as an additional security feature for process creations (see Figure 3).

E. Convert graphs into windows

Graph to sequence. Once provenance graphs are enriched with security features and command-line embeddings, each graph is turned into a sequence of events. For this purpose, we take a full provenance graph, and traverse its nodes in temporal order, to form a sequence of events. This gives EAGLEEYE the following benefits. One, based on a predefined sequence length, we can process a graph as soon as we have that many events. This allows for early detection of malware. Two, sequences can be stored and traversed in an efficient way. For a real-world endpoint detection system, low memory consumption is of high importance. A memory-optimized implementation of EAGLEEYE needs to keep only two data structures in memory: a continuously updated tree with the process call hierarchy, and a flat list of recent behavior events.

When the graph-to-sequence transformation is applied to the provenance graph of the Raccoon Infostealer (from Figure 1), we get a sequence of fixed-size vectors as shown in Figure 8 (for readability, only three important behavior events are

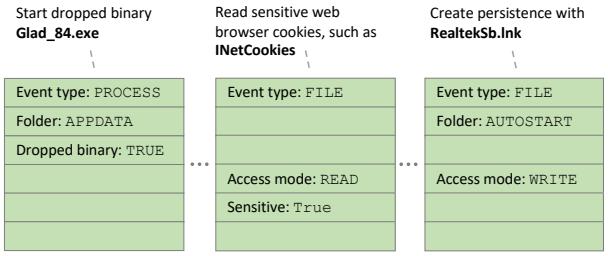


Fig. 8: Sequence of events represented by security features, for Raccoon Infostealer

shown). The resulting sequence consists of security feature vectors, where each vector element is of categorical, numerical, or boolean nature; with the exception of the command-line embedding, which is a 16-dimensional numerical vector. As a case in point, the dropped binary is represented by the following three security features, among others: the event type is `PROCESS`, the executable is located in the user directory `APPDATA`, and the binary is marked as dropped. As mentioned in Section IV-B, the final embedding of all security features is learned during the training phase of the neural network via back-propagation.

During the transformation of a graph to a sequence, it is crucial to keep all security relevant information intact. Since the graph enrichment step uses the whole graph to add rich features to each node, the corresponding sequence represents the behavior of all processes in the provenance graph, and thus contains global information about the relationship between different events.

Sequence to windows. For efficiency, Transformers are typically trained with batches of fixed-size windows. Following this common practice, we slice each event sequence into fixed-size windows. While windows only contain a limited number of *local* events, they still contain *global* information from the original provenance graph. This is due to the fact that graph enrichment happens on a global level (see Appendix F).

For defining windows, we face an inherent trade-off between detection speed, which is required for a real-time detection system, and high malware detection accuracy. On one hand, it is imperative to use windows that are long enough, so that malicious patterns manifest themselves within one window. On the other hand, a detection system is supposed to be run on endpoints with minimal resource consumption, and therefore overly long windows are better avoided. Our experiments show a significant gain in accuracy with increasing window size until around 200 (see Appendix B). Beyond this, increasing the window size further does not lead to a significant accuracy improvement worth the additional detection latency; we therefore choose to use windows of 200 behavior events.

One more important design choice remains—the starting point of windows. A naive approach slices consecutive windows with neither gaps nor overlaps. However, we find that most malware, e.g., the Infostealers, execute malicious payloads soon after a new process has started. Averaged over both malicious and benign application samples of the REE-2023 dataset (Section VI-A1), process creations make up roughly 1% of all observed behavior events, and they therefore offer a good candidate as window starting points.

F. Implementation of EAGLEEYE

EAGLEEYE consists of the data processing pipeline, which turns system logs into windows of security features, and the Transformer model for window classification. The Transformer model for the REE-2023 dataset (Section VI-A) has a stack of four encoders, eight attention heads, and an internal token dimension of 60. The Transformer is trained from scratch on labeled benign and malicious sequences to perform binary classification. Training is quick and takes 26 minutes on an Nvidia GPU V100, thereby allowing for rapid hyperparameter tuning. See Appendices C and D for more details. The Transformer model is relatively small at a parameter size of 2.3 MB, and is therefore suitable for lightweight inference (see Section VI-C6 for real-time resource measurements). The implementation of the Transformer model in EAGLEEYE as well as the data processing pipeline are available on our GitHub repository. The model, implemented using the TensorFlow library [41], takes only $\sim$200 lines of code.

VI. PERFORMANCE EVALUATION

We describe the datasets used for evaluation (Section VI-A), followed by introducing the state-of-the-art malware detection solutions that we compare against (Section VI-B). The comparison of EAGLEEYE with these baselines is done in Sections VI-C1 and VI-C2. We then analyze i) the capability of the Transformer model (Section VI-C3), ii) the benefit of command-line embeddings in EAGLEEYE (Section VI-C4), and iii) the impact of the security features (Section VI-C5).

A. Datasets

We use two datasets for evaluations; see Table II for an overview.

1) Real-world Enterprise EDR (REE-2023) dataset: This proprietary dataset contains data which we collected from an enterprise environment. To monitor the behavior of both benign and malicious applications, we leverage a commercial EDR tool, which uses ETW [42] and Sysinternals [43] to monitor kernel calls. The complete dataset is 15 GB in size and contains 29.2 million behavior events and 12,700 provenance graphs. The benign part of the dataset was gathered during four months in 2022 and 2023, from four different enterprise users located in three different countries in Asia and Europe. The dataset contains behavior from hundreds of applications running on Windows OS, used by the users while carrying out their daily work. For the malicious part of the dataset, we deployed a safe sandbox environment to execute malware and collect the corresponding data, thus ensuring that no damage is inflicted by malware on real endpoints. All malware samples were downloaded from VirusTotal in 2022 and 2023, are in executable format for Windows, and were executed on a special CAPE sandbox. We worked together with security engineers to tune the CAPE sandbox, to make it hard for the malware to detect the sandbox environment. We performed sanity checks to see if the samples included malicious activities, and that

TABLE II: Overview of datasets

REE-2023 dataset	**Benign**	**Malicious**
Graphs	5,580	7,160
Events	6.14 million	23.1 million
Windows of 200 events	9,140	9,140
DARPA-5D dataset	**Benign**	**Malicious**
Graphs	721	91
Events	130,000	22,800
Windows of 200 events	2,370	2,520

they were not malfunctioning. Since the automatic start of malicious samples in the sandboxing environment always happens in the same way, we do not include the first process start in the provenance graph. Lest class-related artifacts are left in the data, we apply the same logic to benign provenance graphs. On average, the benign and malicious provenance graphs of the REE-2023 dataset have a size of 2,300 behavior events, and the graphs have an average depth of 3.5 nodes. For the REE-2023 dataset, we use four event types, namely: new process creations, file access operations, network connections, and Windows registry events. For the benefit of the research community, we release the malware dataset via our GitHub repository. However, the benign dataset is private and has sensitive user information; for ethical reasons, we do not release the benign dataset.

Ethical concerns: The benign data was collected from the corporate computers of employees used for regular work. This was a voluntary program, and interested users were informed of the details of data collection—what will be collected, the purpose, and the period the data will be retained. No personally identifiable information (PII) of users was stored; any user names in the collected data (e.g., file names) were anonymized before storing. The data is stored securely on a company server, where only a limited group of authorized employees have access to. The retention policy states that the data will be deleted six months after the research phase is completed.

2) DARPA FIVEDIRECTIONS (DARPA-5D) dataset: The Transparent Computing (TC) program of DARPA released multiple public datasets [44] containing system logs from APT attacks (advanced persistent threat). The TC program held engagements number 3 and 5, in 2018 and 2019, respectively. Both engagements consisted of a red team trying to penetrate a target network and exfiltrate sensitive information. Since attacks were carried out manually, the malicious parts of both datasets are limited in size and variability, and we therefore choose to combine both datasets to form one larger dataset. We focus on the attacks tracked by the FIVEDIRECTION team; they targeted hosts running Windows 10. The red team used a variety of attack vectors, including Firefox backdoors and phishing emails with malicious MS Office macros. Labeling of the provenance graphs (as malicious and benign) requires manual work, as the public version of the DARPA TC dataset has no labels attached to behavior events. We use the same labeled dataset as ProvNinja [45], which includes the following behavior events: file creations, process creations, and network socket creations. Details on how we preprocessed and labeled this dataset can be found in Appendix E.

B. Baselines for evaluation

We compare EAGLEEYE with two existing malware detection systems: ProvDetector [23] and DeepCASE [22].

ProvDetector [23] is proposed as an anomaly detection solution that learns to distinguish between common system entity interactions and rare ones. This frequency map is used to find rare paths in a provenance graph, where a series of rare events are connected together. Subsequently, a document embedding [46] is learned on the rare paths. At test time, anomalies in the embedding space are detected as malware. To make a fair comparison with EAGLEEYE, we adapt ProvDetector to perform supervised classification. We do so by replacing the final anomaly detector model with a Random Forest classifier. In this revised setting, ProvDetector learns from both malicious and benign graphs. We contacted the authors, and they helped us re-implement their closed-source solution. As an additional reference, we leveraged an open-source re-implementation of ProvDetector by Goyal et al. [29]. To avoid overfitting to the training data, we use the same abstraction method as in the original paper, and remove user names and root directories from paths of files and executables, and mask out local IP addresses.

We carried out extensive hyperparameter tuning to achieve the best possible results with ProvDetector. On the REE-2023 dataset, we obtain the best validation results with a doc2vec dimension of 100. For classification of path embeddings, we tried both SVMs (support vector machines) and Random Forest classifiers, where the latter model achieved better results. Since the classifier should produce a probability score (so as to obtain the ROC curve), we use the percentage of malicious paths to compute a prediction probability per graph.

Additionally, our analysis reveals that there is large variability in the file paths and process names in REE-2023 dataset. Since ProvDetector relies on the fact that system entities in the training data have the exact same name as in the test data, we apply several measures to remove any adverse effect of the datasets. First, we remove alphanumerical identifiers from file and process names. Second, we remove paths, since they vary significantly between users, and only keep file or process names. Finally, we use 100 rare paths per graph during training, instead of the 20 rare paths proposed in the original paper. These steps helped in obtaining the best results with ProvDetector on the REE-2023 dataset.

DeepCASE [22] is a semi-supervised system which aims at reducing the workload for operators of a SIEM (security information and event management) system. DeepCASE makes use of a GRU with attention model, which is trained to take in a sequence of security events and predict the next event. The attention-weighted GRU states are extracted and passed to a clustering algorithm. An analyst finally reviews a few samples per cluster, to perform labeling in semi-supervised way. Since our goal is to have a fully automated security solution, we use the labels of event sequences, i.e., ground

truth, to automatically assign a label to a cluster, thereby turning it into a supervised solution. We use the open-source implementation [47] of DeepCASE for our experiments.

For achieving good results, we perform several adjustments to the DeepCASE implementation. First, we replace the SGD solver with Adam [48]. Next, we use probabilistic cluster labels, instead of binary ones: malware probabilities are based on the *average* number of malicious samples in a given cluster. Finally, to avoid overfitting, we add weight regularization, in addition to the existing dropout layer. The hyperparameters for DeepCASE are provided in Appendix D. We optimized the configuration for each experiment to achieve the best result. The sequence length for DeepCASE is set to the same as that of EAGLEEYE, that is 200 events; note, this value is much higher than the 10 events used in the original paper [22].

C. Results

For comparison of different solutions on a given dataset, we always use the same training:validation:test split of 60:20:20. We perform hyperparameter tuning for each experiment separately, on the validation dataset; see Appendix D for details. A model's accuracy is reported on the test dataset. Evidently, data seen during training does not appear during testing.

Metrics: For performance evaluations, we use the common metrics of TPR (true positive rate) and FRP (false positive rate), where malware is the positive sample. In practice, the false positives, i.e., benign behavior being classified as malicious, are a burden to the security analysts who have to go through the alerts manually, leading to *alert fatigue*. Therefore, it is important to measure the TPR at low values of FPR. To demonstrate the trade-off between low FPR and high TPR, we plot the ROC (receiver operating characteristic) curves for all experiments. We also report the AUC (area under the curve) and the classification accuracy (percentage of correct predictions made). Note, EAGLEEYE and DeepCASE [22] are sequence classifiers. Whereas, ProvDetector [23] works on whole provenance graphs; therefore, we report ProvDetector results for graph classification.

1) Comparison of EAGLEEYE *with baselines, on REE-2023 dataset:* We now compare EAGLEEYE with the state-of-the-art solutions, namely DeepCASE [22] and ProvDetector [23]. In real-world scenarios, endpoints will be hosting both well-known applications as well as some rare or new applications. For this reason, security solutions should have the capability to learn from existing benign applications, and generalize this behavior to *unseen* applications. To emulate such a scenario, we split the benign part of the REE-2023 dataset by *end user*. Therefore, the test dataset contains behavior data from users whose data is not present in training.

Figure 9a plots the ROC curves of the three solutions, and Table III summarizes the key performance metrics. EAGLEEYE performs better than both the baselines. Specifically, EAGLEEYE detects more than 97% of all malware samples at a low FPR of 10^{-2}. Importantly, EAGLEEYE maintains a high TPR of 94.3% even at a much lower FPR of 10^{-3}. In comparison, ProvDetector shows good detection capability only until $\approx 10^{-1}$ FPR, at which point a further reduction of false positives leads to a sharp decline in malware detection capability.

TABLE III: EAGLEEYE vs. baselines, on REE-2023 dataset

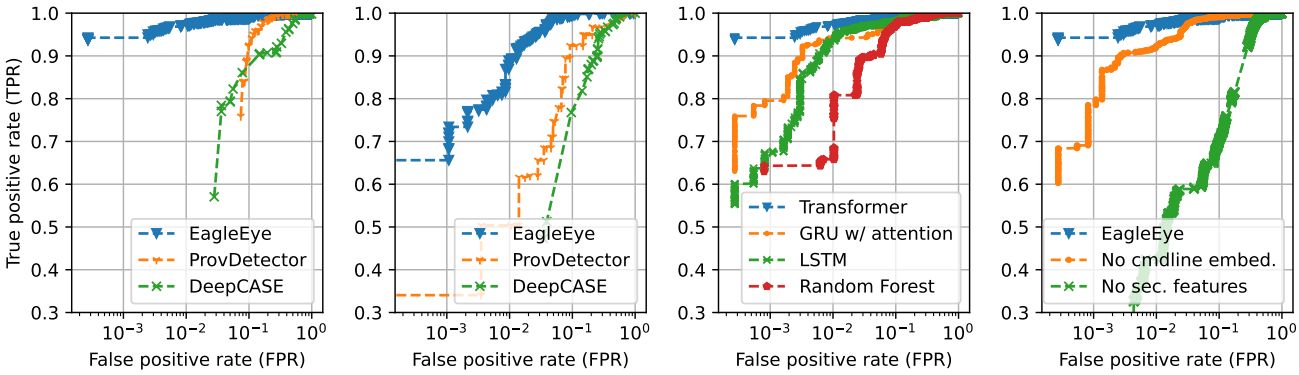
	EAGLEEYE	**ProvDetector**	**DeepCASE**
AUC	99.7%	94.5%	92.5%
TPR at 10^{-1} FPR	98.8%	92.4%	86.1%
TPR at 10^{-2} FPR	97.1%	0%	0%
TPR at 10^{-3} FPR	94.3%	0%	0%
Accuracy	94.8%	90.4%	79.7%

We note that the REE-2023 dataset contains hundreds of application types, whereas the original ProvDetector dataset contains only 23 applications. The fact that system entities have varying names on different systems, and that new applications use new file and process names, explains why ProvDetector has limited classification capability on datasets with large variability.

DeepCASE demonstrates the weakest performance, achieving a test accuracy of only 80%. This solution uses coarse event features, namely event types, which means it does not utilize all available data from provenance graphs. In addition, predicting the next behavior event turns out to be challenging for the DeepCASE model; in comparison to a GRU with attention, a Transformer works better on long sequences of log data (see results in Section VI-C3).

2) Evaluation on DARPA-5D dataset: Next, we evaluate the malware detection capability of EAGLEEYE vs. the baselines on the public DARPA-5D dataset. When compared to REE-2023, the DARPA-5D dataset has several key differences. DARPA-5D contains APT attacks, which span over a longer period of time, whereas the REE-2023 dataset contains mostly malware samples which execute their payload quickly. In other words, the malicious behavior is spread across longer time durations in DARPA-5D. Second, DARPA-5D is a highly imbalanced dataset, whereas REE-2023 contains a similar amount of benign and malicious data. Leveraging all available system logs from DARPA-5D, we get only 91 *malicious* graphs, but 12.3K *benign* graphs (see Table II). To alleviate this imbalance, we first undersample benign graphs. Second, we oversample malicious event sequences for the creation of windows. In order to achieve this oversampling, we drop the requirement that each window must start with a process creation. Note that our oversampling strategy entails that windows can have overlapping events; however, no two windows are identical. After this data rebalancing, we get ≈ 2.4K benign and ≈ 2.5K malicious windows, to evaluate EAGLEEYE and DeepCASE. Oversampling of malicious data works slightly differently for ProvDetector, as the system works on provenance paths, rather than event windows. We duplicate malicious graphs in the following manner: we sample from each graph's rarest paths, randomly pick 70% of the paths, and create a new graph from them. We repeat this process until the dataset is balanced.

As shown in Figure 9b, EAGLEEYE outperforms the baselines; and between the baselines, ProvDetetecor again performs better than DeepCASE. At 10^{-2} FPR, EAGLEEYE



(a) EAGLEEYE vs. baselines on REE-2023

(b) EAGLEEYE vs. baselines on DARPA-5D

(c) Comparing different ML models

(d) Command-line embeddings & security features

Fig. 9: ROC curves for malware detection (with FPR in logarithmic scale)

TABLE IV: EAGLEEYE vs. baselines, on DARPA-5D dataset

	EAGLEEYE	**ProvDetector**	**DeepCASE**
AUC	99.5%	95.1%	92.4%
TPR at 10^{-1} FPR	99.5%	92.4%	76.8%
TPR at 10^{-2} FPR	88.9%	50.4%	0%
TPR at 10^{-3} FPR	65.6%	34.1%	0%
Accuracy	96.5%	84.8%	82.9%

TABLE V: Comparison of ML models, on REE-2023 dataset

	Transformer	**GRU**	**LSTM**	**RF**
AUC	99.7%	99.2%	99.2%	98.5%
TPR at 10^{-1} FPR	98.8%	98.2%	98.1%	97.2%
TPR at 10^{-2} FPR	97.1%	94.2%	93.5%	65.8%
TPR at 10^{-3} FPR	94.3%	79.5%	67.3%	64.3%
Accuracy	94.8%	94.2%	93.3%	79.4%

achieves an accuracy of $\approx$ 89%, which is around 38% more than the next best-performing solution—ProvDetector. Table IV presents the overall results.

3) Importance of the Transformer model: As an ablation study, we keep the data processing pipeline of EAGLEEYE fixed, but use alternative ML models for sequence classification, namely: a GRU with attention, an LSTM (without attention), and a Random Forest. Note, LSTM models have been experimented with in previous works [15], [17], [25]. For example, ATLAS [17] trains an LSTM to classify a sequence as attack or non-attack. For each experiment, we keep the data input and features exactly the same—each model is trained on behavior events from the REE-2023 dataset, where an event is represented by the feature vector described in Section V-C. Figure 9c and Table V present the results.

Importance of Transformer architecture: Among the different ML models, the Transformer architecture performs the best. The advantage of the Transformer architecture becomes apparent when we observe the detection rate at low FPR. At a limit of 10^{-3} FPR, the Transformer-based EAGLEEYE system achieves a detection rate (TPR) of 94.3%, which is significantly higher than the second-best model's TPR of 79.5%. This validates our hypothesis that the Transformer model offers the best capability to extract global patterns in sequence data. Both the GRU and LSTM model have to process behavior events sequentially, whereas the Transformer can look at the whole behavior sequence in one time step.

Importance of event order: As a conventional ML model alternative, we perform an experiment using a Random Forest (RF) classifier. In this experiment, inputs are individual events, not sequences, and labels are taken from the sequence that an event comes from. Therefore, classification is performed on a bag of behavior events without considering the order. In terms of prediction accuracy, the RF model shows significantly lower performance, achieving only 79.4% accuracy, as compared to 93.3% or higher for sequence models. This illustrates that the order of events in a behavior sequence contains valuable information about the intent of an application.

4) Impact of command-line embeddings: The embedding of command-line strings into provenance graphs, and subsequently to sequences, is a novel aspect of EAGLEEYE; therefore, we evaluate it independently in this section. Recall, our security features used for representing process creations are rather coarse-grained (see Table VII). It is for this reason that we add embeddings of command-line strings to the feature vector, so as to encode more nuanced information about process creations. In this experiment, we train a Transformer on the sequences of the REE-2023 dataset, but drop all command-line embeddings from the feature vectors. Figure 9d and Table VI present the results. The plots (with and without command-line embeddings) have an increasing gap in detection rate as the FPR decreases; at a low FPR of 10^{-3}, the absolute TPR increase due to command-line embeddings is 15.8%, demonstrating the value of command-line strings for malware detection.

5) Relevance of security features: As a final ablation study, we assess the benefit of using graph-based security features instead of raw features. While most existing works use raw

TABLE VI: Impact of command-line string embeddings and security features; one at a time

	EAGLEEYE	No cmdline embeddings	No security features
AUC	99.7%	99.5%	91.9%
TPR at 10^{-1} FPR	98.8%	99.1%	70.3%
TPR at 10^{-2} FPR	97.1%	92.2%	42.4%
TPR at 10^{-3} FPR	94.3%	78.5%	15.7%
Accuracy	94.8%	93.5%	81.4%

features like event types or raw file and process names, we enrich graph nodes with a set of features that are relevant for security purposes (see Section V-C). We perform an experiment where we keep the architecture of EAGLEEYE fixed, but ignore all security features from the behavior sequences, and only keep the type of each behavior event. Results on the REE-2023 dataset are reported in Figure 9d and Table VI. At an FPR of 10^{-3}, the detection rate of EAGLEEYE drops from 94.8% to an extremely low 15.7%. This supports the argument that feature engineering at the graph level is required to achieve good prediction accuracy.

6) Resource requirements for real-time malware classification: We now measure the overhead incurred for malware detection, when the endpoint in question has an average number of user applications running. We use a commodity laptop with an Intel 3.6 GHz i7 11^{th} Gen with 8 cores, 32 GB of main memory, but no GPU. We include all steps of our data pipeline in the measurements, from preprocessing of the behavior logs, enrichment of graphs, command-line embedding, one-hot encoding of features, to the final inference with the Transformer model. We execute EAGLEEYE in the background on an endpoint that runs several computer applications concurrently. For real-world conditions, we assume an average number of behavior events based on the REE-2023 dataset. Our results show that EAGLEEYE uses an average CPU load of 2.65% and an average memory utilization of 533 MB, which is acceptable for real-world deployments [12].

We note two important points. i) We assume that behavior events are monitored separately with standard EDR tools; thus we ignore this workload for our experiment. That said, modern EDR tools use various optimization techniques such as minifilter drivers [49] and non-blocking messaging queues to keep resource requirements at a minimum. ii) In our EAGLEEYE implementation, more than 80% of compute time is spent on data normalization and one-hot encoding. Therefore, the data processing step offers room for performance optimization.

VII. CASE STUDY

EAGLEEYE can not only detect malware at high accuracy, but it can also explain *why* an application is malicious; in other words, it offers *interpretability*. To achieve this goal, we leverage the Transformer model's attention mechanism, which learns to put the highest attention on the most suspicious behavior events. As a case study, we revisit the Raccoon Infostealer (Section II), which was not part of the training data. We proceed as follows. First, we query EAGLEEYE with

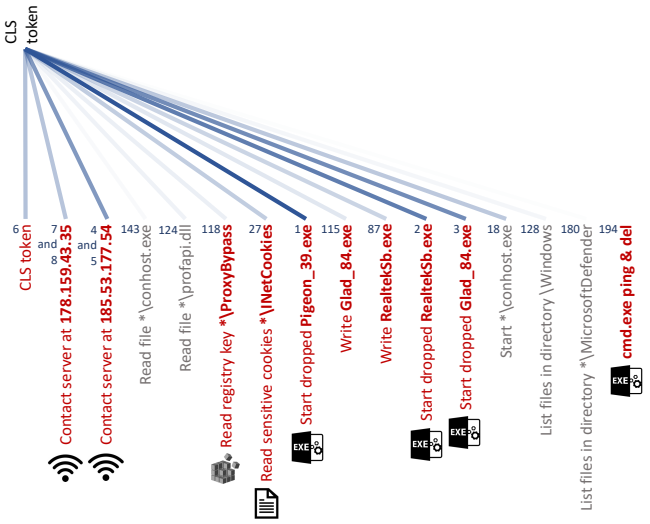


Fig. 10: Attention ranking of EAGLEEYE's Transformer on Raccoon Infostealer behavior events. The lines connected to the CLS token are labeled with the attention ranks. Suspicious events are shown in red.

the malware behavior sequence. Then, we take an attention head from the last layer of the Transformer's attention stack (refer to Figure 5). Next, we measure the normalized attention scores from the representation of the CLS token, the rationale being that, EAGLEEYE's classification head applies only to the final CLS token representation (not the whole sequence).

We report the attention head scores in Figure 10. The figure presents the top 10 *suspicious* events that got the highest attention scores, along with 5 randomly picked normal events, all ordered by time. Each event's attention is shown as a line to the CLS token, where the attention score is depicted via darkness of the line.

Observe from Figure 10 that the starting of three dropped executables are the events most attended to. The next most important events are two outgoing Internet connections and the CLS token representation from the previous encoder layer. We performed a lookup on VirusTotal for the two IP addresses, which confirmed that they have been used as C&C servers during the time. We note that each C&C server is contacted via two distinct socket events (hence the corresponding lines are labeled with two ranks), but for readability, each event is only shown once. Although the last event, `cmd.exe` with self-deletion, is clearly a suspicious one, it did not get significant attention. A potential reason is that the REE-2023 malware dataset does not contain many samples where malware abuses existing executables on the target host. By adding more so-called Living-off-the-land examples to the dataset, the model for command-line embedding would potentially learn to better represent such commands.

We highlight that, based on the attention score, 7 of the top-8 highest-ranked events are indeed *suspicious* events. The only exception is the CLS token representation, as should be expected. Thus, presenting the top-10 events ranked by the attention scores from EAGLEEYE would help a security analyst to quickly interpret a malware detection alarm; e.g.,

if most of the top-10 events look benign, the security analyst can discard the result as a false positive.

As the Transformer model is queried with a sequence of 200 behavior events, its capability to detect the seven most malicious events is promising, thereby demonstrating the capability to extract malicious patterns across long sequences.

VIII. Related work

Over the years, cyber security systems have evolved from basic antivirus solutions that search hashes of known malware, to sophisticated EDR solutions [50] monitoring an extensive list of low-level system events, such as process creations, access to the file system, registry modifications, etc. This fine-grained information is an advantage that provides the much-needed visibility to track process behavior [11]. Hence, a common approach for malware detection on endpoints is to formulate behavior rules, which can then be matched against the collected system logs (e.g., see [51], [18], [52], [53]). However, the logs also pose a challenge due to the vast scale of the collected data, motivating the need for ML models. There exist multiple works that train anomaly detection models on log data [54], [15], [55], [56]. For example, DeepLog learns the behavior of events from recorded history of an endpoint and predicts future events with the goal of detecting anomalies [15].

An important recent advancement is the representation of system events via provenance graphs, thereby helping to capture and present the inherent relationships between the different events [11], [12]. These visual graphs are useful not only for analysts in investigating and interpreting application behavior, but also for developing new security solutions. Based on provenance graphs, recent works looked into providing a high-level abstraction of the low-level events [19], [52], [57], investigating incidents of threats and attacks [17], [22], and detecting anomalies [58], [19], [23], [30], [24].

Broadly, existing works have the following limitations: either the information they extract as features is limited, or the ML algorithm used to train a model from rich and dependent events falls short in learning the context of suspicious events. Earlier solutions, e.g., StreamSpot [58] and Unicorn [20], deconstruct the graph into smaller structures of a fixed size, using embedding functions like StreamHash [58] and HistoSketch [59], to subsequently cluster the embeddings using appropriate distance functions for detecting anomalies. As illustrated in Section II, encoding only graph structure is insufficient for attack detection, and evasion becomes easy [29], [45]. On the other hand, solutions like Sierra [56] do not capture sequential dependency of behavior of events; besides, by using only network logs coming from firewall and IDS systems, protection systems lack the nuanced low-level information that would be available at endpoints.

ProvDetector [23] finds rare sequences of events in provenance graphs and applies the doc2vec embedding on these paths. Our work uses a more advanced event embedding, which is *learned* during Transformer training and uses all available context, not just the adjacent behavior events. As shown in our experiments in Section VI-C1, benign applications may contain rare file and process names. ProvDetector is based on the assumption that all anomalies are malicious, which leads to false alarms in the presence of new applications or changing user behavior.

Other research works have proposed sequence models that are better suited for learning application behavior or application patterns [25], [15], [17], [22]. DeepLog [15] trains an LSTM model to detect, both, execution path anomalies and parameter value anomalies. Atlas [17], developed for constructing attack patterns from a given graph, uses an LSTM model as well. The goal of this work is to aid in attack investigation, and therefore it relies on a threat alert from another source. DeepCASE uses GRU with attention, and trains the system in a semi-supervised way, to provide the context of behavior from event sequences. As observed in our experiments, DeepCASE has limitations in detecting malware with high accuracy, not only because it does not consider security-relevant fine-grained information (Sections VI-C1 and VI-C2), but also due to the inability of the GRU model to learn from long sequential data (Section VI-C3). A similar performance bottleneck is also observed with LSTM models (Section VI-C3).

Recent works have started to explore GNNs (Graph Neural Networks) [35], [30], [21]. For example, ShadeWatcher [21] proposes to use a GNN-based recommendation approach to detect malicious interacting entities (edges) on a graph built from security logs. However, a graph in ShadeWatcher carries limited information, and consequently also does not provide contextual interpretation [60]. Besides, solutions which use the entire graph have a detection lag as they have to wait until the graph is constructed. Similar challenges face ThreaTrace [30]. While promising, GNN models require large datasets to learn effectively from sparse provenance graphs.

IX. Conclusion

To address the challenges in endpoint security, we present EagleEye, a novel system based on a Transformer model. We demonstrate that EagleEye can effectively learn the context of individual actions in long event sequences. Our solution uses rich security features, including a representation of command-line strings which captures the process tree hierarchy. Our evaluation on two datasets demonstrates EagleEye's capability in achieving high detection rates at low FPR. Moreover, our system provides an explanation for malicious behavior via the attention scores of the Transformer model. We publish the implementation of EagleEye and encourage researchers to use the source code as a starting point to make further enhancements.

Given the potential of large language models (LLMs) in cyber security [61], [62], and the inherent language capability of LLMs, a promising next step is to integrate LLMs in endpoint security solutions. Foundational models for security [63] promise to be adaptable to multiple downstream tasks (triaging, malware detection, etc.), while requiring fewer labeled data, and simultaneously providing explainability.

REFERENCES

[1] (Last accessed: July 2024) AV-Atlas. [Online]. Available: https://portal.av-atlas.org/

[2] Statista. (Last accessed: July 2024) Estimated cost of cybercrime worldwide 2017-2028. [Online]. Available: https://www.statista.com/forecasts/1280009/cost-cybercrime-worldwide.

[3] L. Bilge, D. Balzarotti, W. Robertson, E. Kirda, and C. Kruegel, "Disclosure: Detecting Botnet Command and Control Servers through Large-Scale NetFlow Analysis," in *ACSAC*, 2012, p. 129–138.

[4] B. Anderson and D. McGrew, "Machine learning for encrypted malware traffic classification: Accounting for noisy labels and non-stationarity," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, p. 1723–1732.

[5] D. M. Divakaran, K. W. Fok, I. Nevat, and V. L. Thing, "Evidence gathering for network security and forensics," *Digital Investigation (DFRWS Europe)*, vol. 20, pp. S56–S65, 2017.

[6] I. Nevat, D. M. Divakaran, S. G. Nagarajan, P. Zhang, L. Su, L. L. Ko, and V. L. L. Thing, "Anomaly Detection and Attribution in Networks With Temporally Correlated Traffic," *IEEE/ACM Transactions on Networking*, vol. 26, no. 1, pp. 131–144, Feb 2018.

[7] Q. P. Nguyen, K. W. Lim, D. M. Divakaran, K. H. Low, and M. C. Chan, "GEE: A Gradient-based Explainable Variational Autoencoder for Network Anomaly Detection," in *IEEE Conf. on Communications and Network Security (IEEE CNS)*, Jun. 2019.

[8] CrowdStrike. (Last accessed: July 2024) CrowdStrike Falcon. [Online]. Available: https://www.crowdstrike.com/falcon-platform/

[9] SentinelOne. (Last accessed: July 2024) SentinelOne Singularity. [Online]. Available: https://www.sentinelone.com/platform/

[10] T. Micro. (Last accessed: July 2024) Trend Micro Apex One. [Online]. Available: https://resources.trendmicro.com/Apex-One-Upgrade_SE.html

[11] W. Ul Hassan, M. A. Noureddine, D. Pubali, and A. Bates, "OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis," in *Network and Distributed System Security (NDSS) Symposium*, 2020.

[12] F. Dong, S. Li, P. Jiang, D. Li, H. Wang, L. Huang, X. Xiao, J. Chen, X. Luo, Y. Guo, and X. Chen, "Are we there yet? An Industrial Viewpoint on Provenance-based Endpoint Detection and Response Tools," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[13] M. A. Inam, Y. Chen, A. Goyal, J. Liu, J. Mink, N. Michael, S. Gaur, A. Bates, and W. U. Hassan, "SoK: History is a Vast Early Warning System: Auditing the Provenance of System Intrusions," in *IEEE Symposium on Security and Privacy (S&P)*, 2023, pp. 2620–2638.

[14] The Hacker News. (Last accessed: July 2024) New AI Tool 'FraudGPT' Emerges, Tailored for Sophisticated Attacks. [Online]. Available: https://thehackernews.com/2023/07/new-ai-tool-fraudgpt-emerges-tailored.html

[15] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *ACM Conference on Computer and Communications Security (CCS)*, 2017, pp. 1285–1298.

[16] Y. Shen, E. Mariconti, P. A. Vervier, and G. Stringhini, "Tiresias: Predicting Security Events Through Deep Learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, p. 592–605.

[17] A. Alsaheel, Y. Nan, S. Ma, L. Yu, G. Walkup, Z. B. Celik, X. Zhang, and D. Xu, "ATLAS: A Sequence-based Learning Approach for Attack Investigation," in *USENIX Security Symposium*, 2021, pp. 3005–3022.

[18] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "HOLMES: Real-Time APT Detection through Correlation of Suspicious Information Flows," in *IEEE Symposium on Security and Privacy (S&P)*, 2019, pp. 1137–1152.

[19] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage," in *Network and Distributed System Security (NDSS) Symposium*, 2019.

[20] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats," in *Network and Distributed System Security (NDSS) Symposium*, 2020.

[21] J. Zengy, X. Wang, J. Liu, Y. Chen, Z. Liang, T.-S. Chua, and Z. L. Chua, "SHADEWATCHER: Recommendation-guided Cyber Threat Analysis using System Audit Records," in *IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 489–506.

[22] T. Van Ede, H. Aghakhani, N. Spahn, R. Bortolameotti, M. Cova, A. Continella, M. van Steen, A. Peter, C. Kruegel, and G. Vigna, "DEEPCASE: Semi-Supervised Contextual Analysis of Security Events," in *IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 522–539.

[23] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. A. Gunter, and H. Chen, "You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis," in *Network and Distributed System Security (NDSS) Symposium*, 2020.

[24] F. Yang, J. Xu, C. Xiong, Z. Li, and K. Zhang, "PROGRAPHER: An Anomaly Detection System based on Provenance Graph Embedding," in *USENIX Security Symposium*, 2023, pp. 4355–4372.

[25] M. Villarreal-Vasquez, G. M. Howard, S. Dube, and B. Bhargava, "Hunting for insider threats using lstm-based anomaly detection," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[26] Darktrace. (Last accessed: July 2024) The resurgence of the raccoon: Steps of a Raccoon Stealer v2 Infection (Part 2). Https://darktrace.com/blog/the-resurgence-of-the-raccoon-steps-of-a-raccoon-stealer-v2-infection-part-2.

[27] VirusTotal. (Last accessed: July 2024) Internet security, file and URL analyzer. [Online]. Available: https://www.virustotal.com/

[28] (Last accessed: July 2024) MITRE ATT&CK framework. [Online]. Available: https://attack.mitre.org/

[29] A. Goyal, X. Han, G. Wang, and A. Bates, "Sometimes, you aren't what you do: Mimicry attacks against provenance graph host intrusion detection systems," in *Network and Distributed System Security (NDSS) Symposium*, 2023.

[30] S. Wang, Z. Wang, T. Zhou, H. Sun, X. Yin, D. Han, H. Zhang, X. Shi, and J. Yang, "THREATRACE: Detecting and Tracing Host-Based Threats in Node Level Through Provenance Graph Learning," *IEEE Transactions on Information Forensics and Security*, pp. 3972–3987, 2022.

[31] T. Ongun, J. W. Stokes, J. B. Or, K. Tian, F. Tajaddodianfar, J. Neil, C. Seifert, A. Oprea, and J. C. Platt, "Living-off-the-land command detection using active learning," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021, pp. 442–455.

[32] F. Barr-Smith, X. Ugarte-Pedrero, M. Graziano, R. Spolaor, and I. Martinovic, "Survivalism: Systematic Analysis of Windows Malware Living-Off-The-Land," in *IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1557–1574.

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in Neural Information Processing Systems (NIPS)*, vol. 30, 2017.

[34] I. Turc, M.-W. Chang, K. Lee, and K. Toutanova, "Well-Read Students Learn Better: On the Importance of Pre-training Compact Models," *arXiv preprint arXiv:1908.08962*, 2019.

[35] X. Han, X. Yu, T. Pasquier, D. Li, J. Rhee, J. Mickens, M. Seltzer, and H. Chen, "SIGL: Securing Software Installations Through Deep Graph Learning," in *USENIX Security Symposium*, 2021, pp. 2345–2362.

[36] B. Filar and D. French, "ProblemChild: Discovering Anomalous Patterns based on Parent-Child Process Relationships," *arXiv preprint arXiv:2008.04676*, 2020.

[37] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," *arXiv preprint arXiv:1301.3781*, 2013.

[38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of North American Chapter of the Association for Computational Linguistics - Human Language Technologies (NAACL-HLT)*, 2019.

[39] J. Lee, F. Tang, P. Ye, F. Abbasi, P. Hay, and D. M. Divakaran, "D-Fence: A Flexible, Efficient, and Comprehensive Phishing Email Detection System," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021.

[40] H. Face. (Last accessed: July 2024) all-MiniLM-L6-v2. [Online]. Available: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

[41] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/

[42] Microsoft. (Last accessed: July 2024) Event Tracing for Windows (ETW). [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-

[43] M. Russinovich. (Last accessed: July 2024) Sysinternals. [Online]. Available: https://learn.microsoft.com/en-us/sysinternals/
[44] D. A. R. P. Agency. (Last accessed: July 2024) Transparent Computing Engagement. [Online]. Available: https://github.com/darpa-i2o/Transparent-Computing/blob/master/README.md
[45] K. Mukherjee, J. Wiedemeier, T. Wang, J. Wei, F. Chen, M. Kim, M. Kantarcioglu, and K. Jee, "Evading Provenance-Based ML Detectors with Adversarial System Actions," in *USENIX Security Symposium*, 2023, pp. 1199–1216.
[46] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International Conference on Machine Learning (ICML)*, 2014, pp. 1188–1196.
[47] T. van Ede. (Last accessed: July 2024) DeepCASE implementation by authors. [Online]. Available: https://github.com/Thijsvanede/DeepCASE
[48] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference for Learning Representations (ICLR)*, 2015.
[49] (Last accessed: July 2024) Minifilter: Filter Manager Concepts. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/filter-manager-concepts
[50] Gartner. (Last accessed: July 2024) Endpoint Detection and Response (EDR) Solutions. [Online]. Available: https://www.gartner.com/reviews/market/endpoint-detection-and-response-solutions
[51] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. Venkatakrishnan, "SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data," in *USENIX Security Symposium*, 2017, pp. 487–504.
[52] W. U. Hassan, A. Bates, and D. Marino, "Tactical Provenance Analysis for Endpoint Detection and Response Systems," in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 1172–1189.
[53] H. Seo and M. Yoon, "Generative Intrusion Detection and Prevention on Data Stream," in *USENIX Security Symposium*, 2023, pp. 4319–4335.
[54] Y. Yuan, S. S. Adhatarao, M. Lin, Y. Yuan, Z. Liu, and X. Fu, "ADA: Adaptive Deep Log Anomaly Detector," in *Proc. IEEE INFOCOM*, 2020, pp. 2449–2458.
[55] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs," in *International Joint Conferences on Artificial Intelligence (IJCAI)*, 2019, pp. 4739–4745.
[56] J. Lee, F. Tang, P. M. Thet, D. Yeoh, M. Rybczynski, and D. M. Divakaran, "SIERRA: Ranking Anomalous Activities in Enterprise Networks," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2022.
[57] J. Zeng, Z. L. Chua, Y. Chen, K. Ji, Z. Liang, and J. Mao, "WATSON: Abstracting Behaviors from Audit Logs via Aggregation of Contextual Semantics," in *Network and Distributed System Security (NDSS) Symposium*, 2021.
[58] E. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast memory-efficient anomaly detection in streaming heterogeneous graphs," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016, pp. 1035–1044.
[59] D. Yang, B. Li, L. Rettig, and P. Cudré-Mauroux, "HistoSketch: Fast Similarity-Preserving Sketching of Streaming Histograms with Concept Drift," in *IEEE International Conference on Data Mining (ICDM)*, 2017, pp. 545–554.
[60] J. Xu, X. Shu, and Z. Li, "Understanding and Bridging the Gap Between Unsupervised Network Representation Learning and Security Analytics," in *IEEE Symposium on Security and Privacy (SP)*, 2024.
[61] D. M. Divakaran and S. T. Peddinti, "LLMs for Cyber Security: New Opportunities," 2024. [Online]. Available: https://arxiv.org/abs/2404.11338
[62] J. Lee, P. Lim, B. Hooi, and D. M. Divakaran, "Multimodal Large Language Models for Phishing Webpage Detection and Identification," in *Symposium on Electronic Crime Research (eCrime 2024)*, 2024.
[63] M. Sharif, P. Datta, A. Riddle, K. Westfall, A. Bates, V. Ganti, M. Lentz, and D. Ott, "DrSec: Flexible Distributed Representations for Efficient Endpoint Security," in *IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 145–145.

APPENDIX

A SECURITY FEATURES

EAGLEEYE uses a novel feature engineering approach to achieve superior malware detection performance. The *raw* features of each action in the process provenance graph are used to compute structured *security* features. Table VII presents a list of all security features used by EAGLEEYE.

TABLE VII: Security features defined in EAGLEEYE

Event type	Security feature	Feature type
Process start	Name of executable	Categorical
	Path of executable	Categorical
	File extension of executable	Categorical
	Dropped binary	Boolean
	Length of command-line string	Numerical
	Number of command-line flags	Numerical
	Command-line embedding	Numerical vector
File access	File path	Categorical
	File extension	Categorical
	Access mode (w/r/d)	Categorical
	File access options	Categorical
	Sensitive file	Boolean
	Access amount	Numerical
Registry access	Internet key	Boolean
	Persistence key	Boolean
	Uninstall key	Boolean
	Notify key	Boolean
	Data type of key	Categorical
Network connection	Is source internal	Boolean
	Is destination internal	Boolean
	Service port	Categorical
	Connection size	Numerical
	Transport layer protocol	Categorical
	Incoming or outgoing	Boolean
All events	Time duration	Numerical

B TRADE-OFF BETWEEN WINDOW SIZE AND ACCURACY

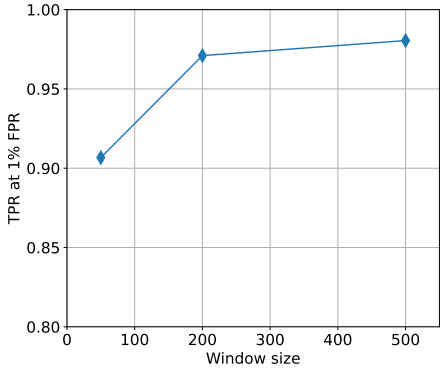


Fig. 11: Trade-off between window size and classification accuracy

EAGLEEYE uses sequences of behavior events as its input. Longer sequences contain more information and thus lead to better classification accuracy. However, short sequences offer the benefit of faster malware detection, as well as lower resource requirements for real-time malware protection on endpoints.

In Figure 11, we present the malware detection rate for different window sizes, specifically at an FPR limit of 10^{-2}. For a window size of 200, the TPR is 97.1%. When we reduce the window size to 50, TPR drops to 90.7%, which is a significant loss in detection capability. When we increase the window size from 200 to 500, we observe a marginal TPR increase of only 1%. Therefore, we select a window size of 200 for our evaluation in Section VI, which strikes a good balance between detection latency and classification accuracy.

C Training progress of EagleEye's Transformer

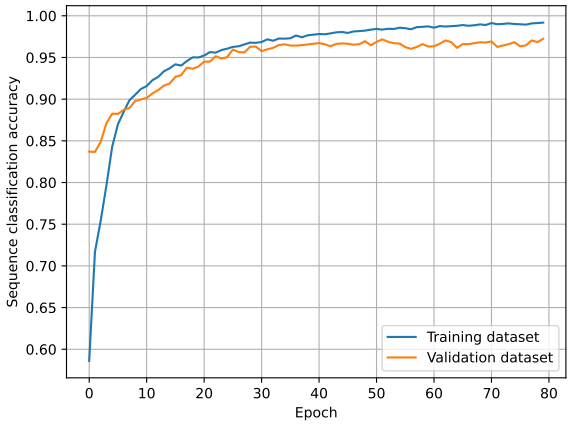


Fig. 12: Training of EagleEye's Transformer on the REE-2023 dataset

EagleEye uses a Transformer model to detect malicious event sequences. The training progress of the best-performing model on the REE-2023 dataset is shown in Figure 12. The training accuracy continuously rises for the first 80 epochs. The validation accuracy lags slightly behind the training accuracy, which is probably due to a slightly more challenging validation dataset. While training accuracy increases till the end, the validation accuracy doesn't grow significantly after 40 epochs. Our hyperparameter tuning shows that a low learning rate is crucial for achieving a solution that generalizes well to unseen data.

D Hyperparameters for machine learning models

In this section, we give more details about the hyperparameters of all the machine learning models used during our evaluation. We include architecture details for EagleEye, as well as the baselines, ProvDetector [23], and DeepCASE [22].

Table VIII shows hyperparameters of EagleEye's best Transformer model. The Transformer model takes in as input, a sequence of behavior events represented by security features, and returns a malware prediction probability. In the table, *Token dimension* refers to the dimensionality of the vector used to represent the hidden state of each token in the encoder stack. *Dimension in FF* refers to the dimension of the hidden layer in the feed-forward neural network, which is located between multi-head self-attention layers. L2 Weight regularization is applied to the dense layers in the classification head. Dropout is applied in several places: after token embedding, in the multi-head self-attention layers, in the feed-forward neural network, and in the classification head's dense layer.

TABLE VIII: Hyperparameters and model size for best EagleEye Transformer

	REE-2023	DARPA-5D
Total parameter size	2.3 MB	1.3 MB
Training time for all epochs	26 min	28 min
Number of encoders	4	6
Number of attention heads	8	6
Token dimension	60	40
Training epochs	80	140
Learning rate	10^{-5}	$5 * 10^{-5}$
Dropout	0.1	0.1
Weight regularization	10^{-2}	10^{-2}
Dimension of attention keys	60	40
Dimension in FF	240	160
Batch size	64	4

TABLE IX: Hyperparameters and model size for best ProvDetector classifier

	REE-2023	DARPA-5D
Rare paths (train)	100	100
Rare paths (test)	20	20
Maximal path length	10	10
Doc2vec model size	238 MB	3 MB
Doc2vec embedding dimension	100	20
Doc2vec training epochs	50	100
Random Forest model size	5.3 GB	1.37 GB
Random Forest trees	2000	2000
Maximal tree depth	20	20

Next, Table IX shows the hyperparameters for our best ProvDetector implementation, on both the REE-2023 dataset and DARPA-5D. We provide numbers for the doc2vec model, which is responsible for embedding rare sentences of varying length into a fixed-size vector representation. Moreover, we show details of the Random Forest model, which classifies the fixed-size vectors into benign and malicious. *Maximal path length* refers to the maximal length for rare paths. Longer paths are cut into multiple parts.

TABLE X: Hyperparameters and model size for best DeepCase classifier

	REE-2023	DARPA-5D
Model size	5.4 MB	4.6 MB
Sequence length	200	200
Training epochs	3	5
GRU hidden nodes	64	64
Epsilon for DB Scan	10^{-2}	$2 * 10^{-2}$
Minimum cluster size	5	3
Weight regularization	10^{-2}	10^{-2}
Learning rate	$5 * 10^{-3}$	10^{-4}
Confidence threshold	0	0
Solver	Adam	Adam
Cluster labels	probabilistic	probabilistic

Finally, Table X lists hyperparameters for the best DeepCASE models on REE-2023 and DARPA-5D. Weight regularization was applied to the GRU sequence model. The confidence threshold was set to zero, to ensure we always get a prediction.

E DARPA-5D DATASET

The DARPA TC program released system log tracing data for engagements number 3 and 5. During engagement 3, attackers used Firefox to download and execute a backdoor named DRAKON as part of the APT group's toolset; moreover attackers used phishing emails with malicious MS Office macros to install malware beacons. For engagement 5, attackers used a Firefox backdoor to download and execute the payload DRAKON and MICRO backdoor.

Since the official DARPA dataset is not labeled at the event level, we describe here how we performed labeling, preprocessing of the data, and provenance graph creation. We thank the authors of ProvNinja [45] for providing us with a labeled version of the DARPA TC-3 and TC-5 dataset. For creation of labeled graphs, the publicly available red team reports are leveraged. With help of the reports, malicious system entities used during an attack are located, and the entities are used as starting points to grow malicious provenance graphs. The growing of the provenance graphs is done via breadth-first search with a maximal depth of 8. Malicious system entities can include malicious URLs, sensitive files, abused living-off-the-land binaries, and dropped binaries.

Benign behavior happens concurrently with attacks. Such behavior includes normal web browsing, use of Office applications to create documents, and reading of emails. Note that benign and malicious provenance graphs can overlap in time. Once malicious graphs are created, all remaining unused behavior events are leveraged for the creation of benign graphs. Attacks in the TC program start with either Firefox or Microsoft Excel. Both applications also get used regularly for benign activity. To make the malware detection task as realistic as possible, both benign and malicious graphs contain at least one Firefox application; additionally, most graphs also contain a Microsoft Excel application.

Note, our labeled version of the dataset might differ slightly from other works, as separating benign from malicious activity is a manual process and entails various design decisions. In particular, our dataset consists of *provenance graphs*, where each node has exactly one incoming edge, namely from its parent process.

F ENRICH EVENTS WITH GLOBAL CONTEXT

EAGLEEYE runs inference on windows of 200 events to detect malware. While event windows are of limited size, they first go through an enrichment phase and thus contain global information from the whole provenance graph. A resource-efficient implementation necessitates a global context which stores the process call hierarchy as a tree, as well as a list of persisted binaries. This global data structure allows us to discard old behavior events after a grace period.

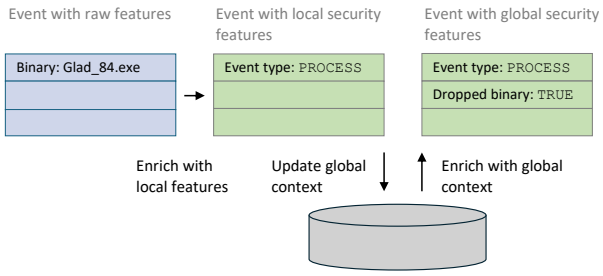


Fig. 13: EAGLEEYE uses global context from the process provenance graph for computation of security features.

Figure 13 depicts the use of both a local and global context. As a case in point, the figure shows a behavior event which represents the start of a binary called `Glad_84.exe` from the previously discussed Raccoon Infostealer. For each event, *local* raw features are used to compute *local* security features. In this case, the system can infer the event type as `PROCESS`. Next, the global context is queried to infer global security features. The fact that this binary was dropped might not be visible in this or the neighboring behavior events. Instead, the global data structure contains the necessary information that a binary with exactly this name and path has been written to disk before. As a final step, the global context is updated with the raw information of this particular event. In this case, the call hierarchy tree is extended by this process. In the end, the event is represented by all security features, and now contains both local and global information.