2024 Annual Computer Security Applications Conference (ACSAC) | 979-8-3315-2088-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/ACSAC63791.2024.00072

DEEPCAPA: Identifying Malicious Capabilities in Windows Malware

Saastha Vasan*, Hojjat Aghakhani*, Stefano Ortolani†, Roman Vasilenko‡, Ilya Grishchenko*, Christopher Kruegel*, Giovanni Vigna*
*University of California, Santa Barbara, †Broadcom, ‡Google
{saastha, hojjat, grishchenko, chris, vigna}@ucsb.edu, ortolanis@broadcom.com, rvasilenko@google.com

***Abstract*—Malware detection and classification has been the focus of extensive research over many years. However, less effort has been devoted to developing post-detection systems that identify specific malicious capabilities (or behaviors) in malware. Such systems play a critical part in identifying and mitigating the damage caused by malware attacks. Unfortunately, current methods for identifying malware capabilities involve substantial manual reverse engineering efforts and context switching between multiple tools, which slows down an investigation and gives attackers an advantage.**

In this paper, we propose DEEPCAPA, an automated post-detection system that uses machine learning to identify potentially malicious capabilities in malware in the form of MITRE ATT&CK techniques. Our system operates on sequences of API calls, extracted from the memory snapshots taken at key points during the (sandboxed) execution of malware. Our results demonstrate that DEEPCAPA can accurately identify malicious capabilities, achieving a precision of 95.80% and a recall of 93.76% across 29 different techniques.

1. Introduction

Malicious software (malware) poses a significant threat to computer networks and users. In recent years, there has been a steep rise in the number of malware attacks. According to statistics published by AV-TEST [1], more than 100 million new malware samples were detected in 2023. Failure to detect and mitigate these threats can be catastrophic for organizations and individuals, resulting in data breaches, financial losses, and damage to reputation.

To combat the ever-increasing number of malware threats, a significant body of research has proposed both static and dynamic techniques to detect malicious code [2]–[6]. More recent approaches have focused on machine learning models, often based on deep neural networks [7]–[13]. These models use features that are extracted via static or dynamic analysis techniques, and their goals are either to distinguish between malicious and benign samples [14], [15] or to classify malware into different families, such as ransomware, remote access tools, or information stealers [16]–[18].

Despite achieving good detection rates, most existing methods provide only limited information about particular malicious techniques that a malware sample employs. However, developing effective mitigation strategies relies on detecting these techniques to identify which system components require patching. Consider a sample that is detected as malicious by a malware detection solution. This malware can perform "Registry Modification" to achieve persistence, "Process Injection" to gain root privileges, "C2-Communication" to upload system information, and implement "Anti-VM" techniques to thwart dynamic analysis. Identifying these capabilities requires a deeper understanding of the malware's behavior and a more sophisticated analysis. This is in contrast to malware detection solutions that are designed to focus solely on identifying the presence of malware. Currently, analysts in a Security Operations Center (SOC) are mostly left to identify and understand these malware capabilities manually. This process involves loading the binary into a disassembler to extract static artifacts and combining this information with dynamic artifacts obtained using a debugger as well as network and memory forensics tools. Unsurprisingly, this process is difficult, slow, and error-prone. Even with automated dynamic analysis techniques, malicious capabilities can go undetected if the malware checks for certain environmental conditions before executing its payload, such as the OS version, the presence of certain files, processes, and registry keys.

Recognizing this issue, recent academic work has sought to automate the identification of malware capabilities. For instance, Forecast [19] examines memory snapshots of malware and uses symbolic analysis of its code to predict capabilities. The industry has also recognized the problem, with researchers from Mandiant releasing CAPA [20], a tool that uses static analysis to detect capabilities in executable files in the form of MITRE ATT&CK techniques. The industry relies heavily on CAPA to determine the high-level capability of malware before performing deeper (and more expensive) manual malware analysis.

In this paper, we present DEEPCAPA, an automated *post-detection* system that uses deep neural networks along with features extracted from process dumps of malware programs to identify high-level, potentially malicious behaviors in the form of MITRE ATT&CK techniques. Unlike Forecast and CAPA, which rely on manually crafted rules to describe malicious techniques, our system can learn the rules automatically, even when the code is obfuscated. This approach makes DEEPCAPA more generic, more robust, and less likely to miss relevant behaviors, which reduces the likelihood of false negatives – an observation confirmed by our experimental evaluation results (Section 4). Our system relies on sequences of Windows API and library calls[1] extracted from temporal memory snapshots taken by running malware in a sandbox environment. Unlike CAPA, which cannot analyze packed malware, our feature extraction on dynamic artifacts allows us to capture features from packed code. As a result, our approach intuitively achieves better code coverage. It is also much more likely to include important dormant functionality of the malicious code, which is missed by purely dynamic API call traces.

Our deep learning model operates in two distinct stages. In the first stage, we perform pretraining on sequences of API calls. This pretraining process is crucial, as it allows our model to learn the dependencies between different API calls. This is similar to how language models for text learn the dependencies between words in a sentence [21]–[24]. Following the pretraining step, in the second stage, we utilize transfer learning to fine-tune our models to identify capabilities. We have also incorporated attention layers into the architecture of our fine-tuning step. Attention is a unique mechanism that compensates for the lack of explainability often associated with neural networks. This allows us to extract important sequences of APIs that contribute significantly towards identifying the presence of a MITRE ATT&CK technique.

To train and evaluate DEEPCAPA, we use a dataset of 17,336 samples, along with their process snapshots (this data was provided by a collaborating security vendor). For each sample, we also know the MITRE ATT&CK techniques that were observed during dynamic sandbox execution. In total, DEEPCAPA can detect 29 different MITRE ATT&CK techniques (behaviors).

Our experiments show that our system delivers high precision and recall (95.80% and 93.76%, respectively) across all 29 techniques. We further extend our analysis of DEEPCAPA by performing a baseline comparison with other state-of-the-art malware capability detection frameworks, in particular, Forecast and CAPA. We also manually analyzed the attention results, and reverse-engineered and inspected the disassembly code for a subset of false positives and false negatives.
In summary, we make the following contributions:[2]

- We propose DEEPCAPA, a post-detection system that operates on memory snapshots of malware and identifies capabilities based on the MITRE ATT&CK taxonomy.
- We introduce a feature engineering pipeline to extract API call sequences by processing one or more memory snapshots.
- We designed and implemented a deep learning pipeline that is capable of capturing the relationships between API call sequences and program behaviors and that provides insights to a human analyst by leveraging the attention mechanism of our neural network.
- Our results show that DEEPCAPA delivers accurate results. Moreover, our comparison with state-of-the-art systems demonstrates that our approach to automatically learning behaviors is more general and leads to fewer false negatives.

1. We refer to both Windows library and system calls as "API calls" for simplicity.

2. We will publicly release the source code of our system at https://github.com/ucsb-seclab/DeepCapa

2. The MITRE ATT&CK Framework

The ATT&CK framework was originally created by MITRE [25] out of a need to systematically categorize the ever-changing landscape of malware tactics, techniques, and procedures (TTPs). The MITRE ATT&CK framework is considered a modern way of looking at multi-stage cyber attacks (kill chains) and has become the industry standard for describing malicious behaviors.

At a high level, the MITRE ATT&CK framework is a behavioral model that categorizes attacks into *tactics* and *techniques*. Tactics denote short-term, tactical adversary goals of an attack. They include *Persistence*, *Privilege Escalation*, *Defense Evasion*, *Credential Access*, *Discovery*, *Lateral Movement*, *Execution*, *Collection*, *Exfiltration*, and *Command and Control*. Techniques describe the technical means through which tactical goals are achieved (implemented). For example, an information stealer malware program might first mask itself as a legitimate application to trick the user into executing it. Then, it creates a registry key to achieve persistence, performs checks to detect the presence of a virtual machine, collects information from the system, and eventually sends that collected data to its C&C server using the HTTP protocol. The behaviors of this info stealer malware can be described using the following MITRE ATT&CK techniques: *T1036* (Masquerading), *T1027* (Obfuscated Files or Information), *T1112* (Modify Registry), *T1547* (Boot or Login Autostart Execution), *T1082* (System Information Discovery), *T1497* (Virtualization/Sandbox Evasion), and *T1071* (Application Layer Protocol).

3. Methodology

In this paper, we propose DEEPCAPA, a post-detection system to identify the capabilities of a malware program in the form of techniques defined by the MITRE ATT&CK framework [25]. Our system takes as input one or more memory snapshots taken from the execution of a Windows PE executable in a sandbox environment. It then outputs a set of MITRE ATT&CK techniques that this program implements.

Figure 1 shows an overview of our system. DEEPCAPA operates in two main steps. In the first step, it reconstructs a CFG of the program's code from the memory snapshots

(Section 3.1). Using the CFG, the system extracts API call sequences that the program can potentially execute. In the second step, the API call sequences are encoded and fed into a neural network model, which consists of four transformer encoder layers (Section 3.2 and Section 3.3). The goal of this model is to identify malicious capabilities and the corresponding API calls associated with them.

3.1. API Call Sequences Extraction

In the first step, DEEPCAPA extracts possible API call sequences that the program under analysis might invoke. In this work, we use API call sequences as a way to capture and model the behaviors of (malicious) code. This approach has a long tradition: API call sequences have been used by numerous tools to distinguish between malicious and benign activities [21]–[24]. Also, it makes intuitive sense to use API functions – and their sequences – to characterize program behaviors. After all, many behaviors lead to changes in a program's environment or to "visible" and external effects. This includes changes to files or configuration settings, packets that are sent over the network, code that is injected into another process, or windows that are popped up. All these changes require the invocation of the operating system and Windows library API functions, and our system analyses these invocations. The API call sequences extraction step proceeds in four stages: CFG Construction, CFG Merging, CFG Reduction, and CFG Exploration.

CFG Construction. DEEPCAPA takes as input one or more memory snapshots of a Windows PE program that is dynamically executed in a sandbox. A new memory snapshot is taken whenever one of the following interesting system events occurs: **(1)** Execution of an API call that causes a new process creation or a new file creation. **(2)** Virtual memory execution, meaning that code execution happens outside of the original PE image. **(3)** A change to the original PE image (e.g., because of code unpacking).

We extract all sections marked as executable from the memory snapshots, including segments generated during runtime. Then, to construct the control flow graph (CFG) from these code sections, we recognize function boundaries in them and provide this information, together with the code sections, to a disassembler (we use IDA Pro [26]) via our custom-designed loader. Each node in the generated CFG represents a basic block – a set of instructions that execute sequentially. This method of CFG generation effectively handles various malware behaviors, such as dynamic code allocation, making data sections executable, or employing packing and unpacking techniques. Interestingly, our experiments have not revealed any significant efforts by malware authors to obstruct the extraction of control-flow graphs from the unpacked code extracted during runtime. However, we acknowledge that this might change in the future. In such cases, we may need to employ more advanced techniques or leverage prior work on robust CFG generation approaches [27], [28].

CFG Merging. Our CFG Merging technique reconstructs the control flow graph (CFG) of a program from multiple memory snapshots (taken during this program's execution). Since different snapshots may contain different (parts of) the CFG – recall that malware might unpack parts of its code during run-time – we need to merge the CFGs across snapshots to obtain a more comprehensive and accurate view of the program. To perform this, we leverage the information gathered from the previous step, where we identified function boundaries. We start by considering the functions from the first snapshot and iteratively add the CFGs of any new functions encountered in subsequent snapshots. When we encounter a previously seen function, we compare its CFG in the newer snapshot with the one in the previous snapshot. If the newer snapshot's CFG has additional nodes or edges, we incorporate these changes into our merged CFG. Specifically, we analyze indirect calls in each function and add inter-procedural edges if the target information is present in any of the snapshots' CFGs. By applying CFG Merging, we can create a unified CFG, starting from the first snapshot to the last, and obtain a more complete picture of the program.

CFG Reduction. As mentioned previously, we focus on API call sequences to capture program behaviors. Therefore, some information captured in the CFG is redundant, and we filter it out. In particular, we remove all instructions from basic blocks that are neither jump nor call instructions to an internal function or an external API. Then, we remove all blocks with no instructions from the CFG, while keeping the connectivity of the graph intact. To preserve the connectivity, we follow two rules when removing blocks with no instructions: **(1)** We always keep the start block and the end block of each function. **(2)** For every other block, if a block B does not contain any instruction, we generate edges from all of $B's$ parent blocks in the graph to all its successor blocks.

CFG Exploration. Once the CFG has been simplified, we perform probabilistic random walks over the resulting graph to obtain short, fixed-length sequences of API calls. We start the random walk from a node with low API coverage to achieve better code reach. The intuition is that our approach will capture enough relevant sequences so that our neural network model will identify the behaviors. At the end of the CFG Exploration stage, we extract S sequences of API invocations, where S is the number of probabilistic random walks performed on the sample's CFG, and each sequence consists at most of A API calls. In our experiments, we set S to 350, and A to 20. We provide a detailed explanation of our algorithm in the Appendix E.

Discussion of the API Call Sequences Extraction Approach. An important consideration for the API call sequences extraction approach is that it must be robust against malware authors' attempts to evade analysis and hide relevant behaviors (API call sequences). First, we observe that our approach is not influenced by the modification, reordering, or substitution of non-control-flow instructions, blocks, or functions, since the CFG reduction process already removes these instructions. However, the attacker could insert additional API calls along "dummy" paths that are not actually executed during run-time. This is why we

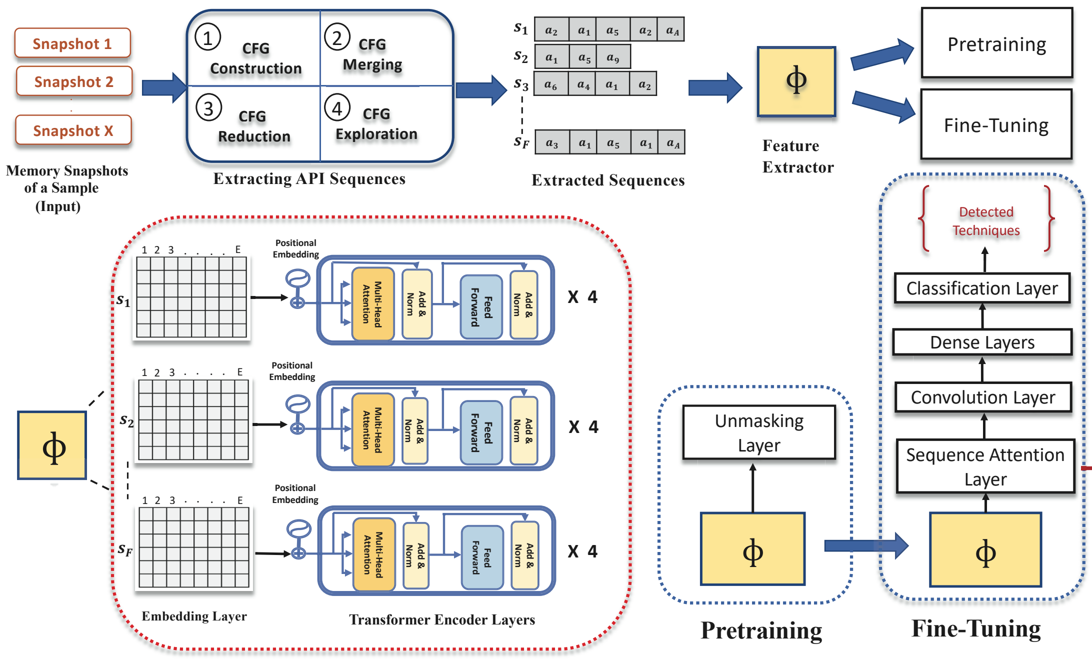

Figure 1: An overview of how DEEPCAPA operates.

perform (many) probabilistic random walks over the graph; the intuition is that our approach will capture enough relevant sequences so that our neural network model will still identify the behaviors. Finally, an attacker could also camouflage relevant API call sequences by adding "padding" between all calls along the execution path, but this padding requires a substantial amount of calls to prevent our model from recognizing the sub-sequence that indicates a behavior. While the goal of DEEPCAPA is not to detect malware, we believe that attackers would not likely expose their malware to easier detection just to thwart the analysis of individual behaviors. We continue our security and robustness discussion of DEEPCAPA in Section 5.

3.2. Neural Network Architecture

In this section, we describe our neural network model. Our model is inspired by the architecture of large language models (LLMs) [29]–[32]. These models use the transformer architecture [33], which was originally proposed to understand the context and meaning of text data by modeling the relationships between words and sentences. We adapt this architecture to the task of detecting malware capabilities based on sequences of API calls.

Our model takes as input a collection of API call sequences $S \times A$ (Section 3.1) of a malware sample x. Each individual API call is treated as a word, and each sequence is viewed as a sentence. Our model views each input sample as a document made up of API call sequences. However, unlike traditional text documents, for which sentence order and position are crucial, the order of sequences within our collection does not affect the program behavior.

We train our model in two phases: pretraining and fine-tuning. In pretraining, we train the model on a large corpus of unlabeled API call sequences to predict an API call given the previous and following API calls within the sequence. This is an unsupervised learning process, where the model learns the general patterns and contexts of the API calls. The pretrained weights of the model (ϕ) capture general information and context from the input data. In fine-tuning, we train the model on a labeled dataset of API call sequences. The labels indicate the malware capabilities associated with each sample. This is a supervised learning process, where the model adapts its pretrained knowledge to the specific task of identifying malware capabilities. We also add an attention layer in this phase, which helps the model focus on the most relevant sequences for detection. The pretraining and fine-tuning phases enable our model to create rich embeddings for the input data, capturing the context and information needed to identify malware capabilities. Prior to explaining these two phases in detail, we first describe how we embed sequences of API calls into numerical representations for processing by the transformer and its subsequent layers.

Feature Extractor Network. Our model's feature extractor network is comprised of three distinct layers. The first layer is the API-call embedding layer. This layer's function is to transform each API call within a sequence into a vector of real numbers representing it in a high-dimensional space. This function enhances the model's comprehension of sequential data and reduces the input size for the subsequent layer. The second layer is the positional embedding layer, which incorporates the position of each API call within the sequence. This helps the transformer models to learn the context and meaning of sequential text data by observing the relationships among them. The third layer is the contextual embedding layer, which consists of transformer encoders that process each API call within a sequence in relation to all others in both forward and backward contexts. This comprehensive understanding of the input data enables the model to capture intricate patterns, dependencies, and relationships within an API call sequence.

3.3. Training Procedure

We now explain the training process of DEEPCAPA consistsing of a pretraining stage, followed by a fine-tuning stage.

Pretraining Stage. We pretrain our model with the Masked Language Modeling (MLM) objective, originally proposed for training Large Language Models (LLMs) [29]–[32]. We randomly mask 15% of the API tokens in each sequence with a special token <*MASK*>, following BERT [30]. We also use four other special tokens: <*SEQ*> and </*SEQ*> to mark the beginning and the end of a sequence, <*PAD*> to fill the sequences with different lengths to a uniform size A, and <*UNK*> to account for out of vocabulary tokens during testing. We then feed the masked sequences to a transformer encoder to capture contextual information. The encoder's output is processed by a layer of size $H \times U$, where H is the embedding dimension, and U is our vocabulary size (Table 1). The goal of the pretraining stage is to uncover the masked tokens and improve the model's understanding of API call semantics and context.

Fine-Tuning Stage. With the API call representations learned from a wealth of unlabelled data and stored within the parameters of the encoder network, we now proceed with the fine-tuning stage. In this stage, we clone the pretrained weights ϕ for each MITRE ATT&CK technique m (i.e., $\phi'_m \leftarrow \phi$), and append it to a new attention layer, followed by a convolution layer, a dense neural network, and a linear *binary* classifier. We will refer to them together as $\widehat{g}_m$. We fine-tune our model $\widehat{g}_m(\phi'_m(x))$ by optimizing the following loss function:

$$\min_{\theta'_m, \widehat{\theta}_m} \mathop{\mathbb{E}}_{(x,y_m)\sim D^{<m>}_{\text{train}}} \left(L\left(\widehat{g}_m(\phi'_m(x)), y_m \right) \right), \tag{1}$$

where θ'_m and $\widehat{\theta}_m$ represent the parameters of ϕ'_m and the linear classifier within $\widehat{g}_m$. The fine-tuning loss function minimizes the cross-entropy between the predicted probabilities and the ground-truth label y_m, which indicates whether the sample x actually implements the technique m or not.

Sequence Attention Layer. Not all API call sequences of a given sample x characterize its behavior equally, so we use an attention network to give weights to each API call sequence in x. When predicting a technique for x, the attention weights can indicate the most influential API call sequences in x.

To compute the attention weights, we first take the mean of the output of the transformer encoder (which has a dimension of $S \times A \times H$) along the dimension A. Doing so gives us a 2-dimensional $S \times H$ array W where each row is the vector representation W_k of the k-th API call sequence created in the transformer embedding space. We then feed W to a shallow neural network with three layers of $H \times H \times 1$ neurons to compute a weight for each sequence vector. We then pass the computed weights to a softmax layer so that the normalized weights sum up to one. We then use these normalized weights to scale the mean vector representation of the API call sequences (W). Therefore, with this attention network, we embed x into a 2-dimensional array (with the dimension of $S \times H$, where rows are sequence hidden vectors). We train this attention network along with the main network during optimization.

Convolutional Neural Network. The input of this layer is a 2-dimensional array where the rows are sequence hidden vectors, each scaled with the corresponding sequence attention weight. Recall that we extract a list of API call sequences for each sample x. The order of the sequences in this list has no effect on the sample's actual behavior, thus, we make use of a one-dimensional CNN. This is because, unlike RNNs, CNNs are better suited to handle shifts in the data. We use a max-pooling layer after multiplying the input with so-called *sliding windows*. We flatten the output of this layer, so it can be processed by a dense neural network.

Dense Network. CNNs are typically followed by a few dense layers, so the CNN features can be processed together by the dense layers prior to the classification. DEEPCAPA follows a similar approach. The result of the dense layer is passed to a dropout layer, which implements dropout randomization [34], a technique that is proposed to prevent neural networks from over-fitting and increase their performance.

Linear Classifier. The output of the dense layer is then passed to a linear classifier, where DEEPCAPA employs a single binary classifier g_m for each MITRE ATT&CK technique m.

4. Experimental Evaluation

In this section, we evaluate and analyze the performance of DEEPCAPA. We assess the general performance of our system using a dataset of 17,336 Windows PE files and 29 unique MITRE ATT&CK techniques. We then perform a comparison with Forecast and CAPA. Finally, we extend our evaluation of DEEPCAPA by manually analyzing false positives and false negatives. To examine if our model is able to generate explainable outputs, we also analyze the attention weights at the sequence call level. Before discussing the results, we first describe in more detail the dataset that we use to train and evaluate DEEPCAPA.

4.1. Dataset

For this research, we received a dataset of malicious samples from a security vendor, along with their memory snapshots and dynamic analysis reports. The dataset consists of 17,336 malicious samples observed in the wild between March 2020 and June 2022. We do not have benign samples

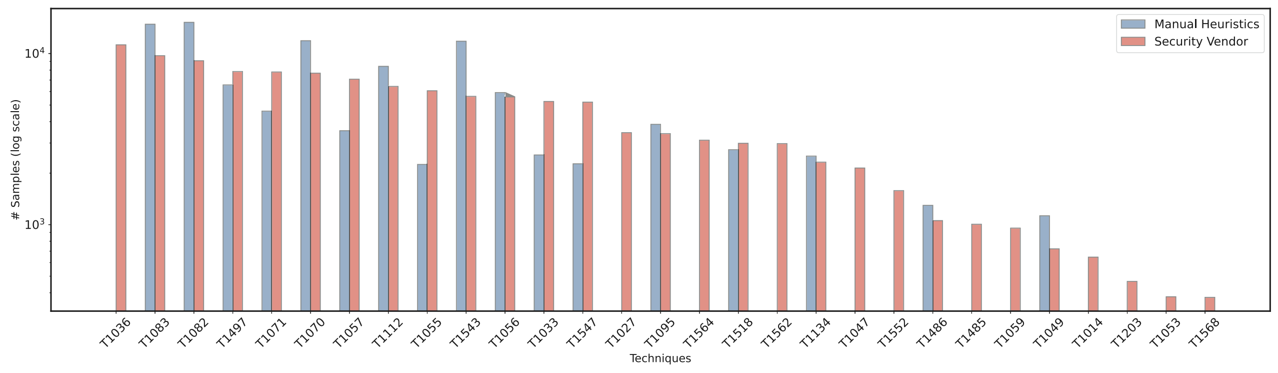

Figure 2: Labels generated by the security vendor (dynamic sandbox reports) and our manual (static) heuristics.

in our dataset, as the goal of DEEPCAPA is not to distinguish between malicious and benign code, but rather to identify the detailed capabilities and behaviors of a sample after it has been detected as malicious by malware detection tools.

To better understand the diversity of malware families in our dataset, we retrieved and analyzed the anti-virus (AV) labels for all samples. We queried the AV labels supplied by VirusTotal [35] and processed them using AVClass2 [36] to extract the malware family names. We observed that AVClass2 was not able to detect malware families for 8.9% of our samples. From the remaining 91.1% of samples, we identified 912 different malware families, covering a wide variety of relevant threats. The ten most prominent families, namely Berbew, Stihat, Virut, Sytro, Stopcrypt, Sivis, Kwampirs, Stop, Formbook, and Noon, collectively accounted for 34.62% of all the samples in our dataset.

Ground-Truth Labels. Training DEEPCAPA requires accurate and comprehensive labels of malware capabilities in the form of MITRE ATT&CK techniques. However, generating these labels is not trivial [37], as malware behavior can vary depending on execution context, environment, and user input. Malware often employs evasion techniques such as encryption, obfuscation, or packing, and can exhibit complex behaviors implemented through various combinations of API calls, making accurate detection and labeling challenging. To address these challenges and obtain comprehensive ground truth labels for our dataset samples, we utilize two sources: dynamic analysis (sandbox) reports from a security vendor and precise (though incomplete) static heuristics we designed.

The security vendor provides a list of MITRE ATT&CK techniques exhibited by the malware sample during sandbox execution. However, this method only captures executed behaviors, potentially missing dormant behaviors, or behaviors present in malware samples employing evasive techniques. To complement the sandbox reports, we developed a set of static API call-based detection heuristics covering 17 MITRE attack techniques. These heuristics aim to capture potential behaviors in both executed and non-executed code and are applied after the CFG Exploration phase (discussed in Section 3.1). Our API call-based detection heuristics are narrow and effective in identifying techniques characterized by specific API calls, such as T1112 (Modify Registry), T1082 (System Information Discovery), and T1049 (System Network Connections Discovery). However, they may miss more complex techniques like T1055 (Process Injection), T1071 (Standard Application Layer Protocol), and T1547 (Boot or Logon Autostart Execution), which require analysis of multiple API calls. Conversely, vendor labels from sandbox reports are more effective in detecting complex techniques but miss those present in dormant code.

We combine static and dynamic sources to compile a more complete and reliable set of malware capability labels. Figure 2 illustrates the number of malicious samples detected for each MITRE ATT&CK technique using both sandbox reports and static heuristics. Across our dataset, we identify 29 unique MITRE ATT&CK techniques, reflecting various phases of an adversary's attack lifecycle.

However, our goal for DEEPCAPA is to be able to generalize when training on these labels, i.e., detect more techniques than the ones provided by the labeling mechanisms. Our analysis in Section 4.4 provides evidence that DEEPCAPA can identify behaviors missed by the labeling methods.

4.2. General Performance

Training Setup. We partition our dataset into training, validation, and test sets with a (70%, 15%, 15%) random split. We use the validation set for hyperparameter tuning and leave the test set *unseen*, except for when we report the results. In particular, our training, validation, and test sets consist of 12,111 samples, 2,573 samples, and 2,652 samples, respectively. As stated in Section 3.3, DEEPCAPA operates in two stages; pretraining and fine-tuning.

For pretraining, we build the feature extractor network ϕ by minimizing the cross-entropy loss function computed over the entire training set. Using the validation set, we tune the hyperparameters of ϕ. Table 1 shows the values we have tried for these hyperparameters and the final selected values.

In the fine-tuning, for each technique m, we use a learning rate of 10^{-4} to optimize the loss function (Equation 1) on a balanced training set, with 50% of the samples having

Table 1: The hyperparameters used to build the feature extractor network ϕ.

Parameter	Candidate Values	Final Value
Batch Size		350
Unique APIs - U		3868
Number of API-call in Sequence - A	[10, 20, 30, …, 50]	20
Number of Sequences - S	[50, 100, 150, …, 400]	350
Embedding Dimension - H	[64, 128, 256, …, 1024]	768
Transformer Encoder Layers -		4
Attention Heads -		4
Dense Layer Neurons - ζ	[16, 32, 64, …, 512]	64
CNN - Kernel Size		(3, H)
CNN - Kernel Stride		1
CNN - Number of Kernels	[3, 4, 8, 16, 32]	8
MaxPool - Kernel Size	[(2,H), …, (7,H)]	(3,H)
MaxPool - Kernel Stride	[1, 2, …, Kernel Size]	1
Dropout Probability	0.2, 0.3, …, 0.9	0.1
Optimizer	[Adam, SGD]	SGD
Optimizer Learning Rate	[0.1, 0.01, 0.001, 0.0001]	0.0001

a positive label (i.e., the technique is present), and 50% of the samples having a negative label (i.e., the technique is not present). To make the results easier to interpret, we use a 50/50 split for the test set as well.

Cost Analysis. Our base system comprises of an Ubuntu 18.04 machine (with 377GB RAM, a 96-core 2.10GHz CPU, and 1xRTX-Titan GPU). We use the NetworkX [38] package to generate CFGs from the memory snapshots, and we leverage the PyTorch [39] framework to implement our neural network model. On average, extracting the API sequences from a sample (consisting of multiple memory snapshots) takes 181 seconds. The training for each batch of input takes 0.12 seconds. For inference, it takes 1.1 seconds to load our model with stored weights and 0.04 seconds to process a batch of inputs and output the detected techniques.

Results. For each MITRE ATT&CK technique m and its corresponding binary classifier $\widehat{g}_m(\phi'_m)$ defined in Section 3.3, we calculate and report the precision and recall for the positive class in Table 9. The Aggregated Precision is calculated as $TP \div (TP+FP)$, and the aggregated recall is calculated as $TP \div (TP+FN)$. Where TP, FP, and FN represent the total number of true positives, false positives, and false negatives, computed over all 29 techniques. As shown in Table 9 in the appendix, the aggregated precision and recall for all techniques are 95.80% and 93.76%, respectively. To interpret these results in an easier way, we divide all 29 techniques into an API-Call-Based Set and a Generic Behavior Set:

API-Call-Based Set (17 Techniques). This set contains techniques that are usually implemented by directly calling one (or a small number of) API calls that are directly responsible for a specific behavior. For example, consider the *T1112: Modify Registry* technique. One common implementation invokes the following sequence of APIs: *RegOpenKey, RegQueryValue, RegSetValue* and *RegCloseKey*. We expect that our classifier recognizes that this sequence is associated with the *T1112: Modify Registry* technique and will report the appropriate behavior when it identifies such a sequence of calls in the code of a sample. Moreover, it is unlikely that this sequence of calls is used for anything other than modifying the registry, so it is unlikely to trigger incorrect results. Since DEEPCAPA bases its detection on (sequences of) API calls, we expect that it will deliver particularly accurate results for techniques in this set.

Generic Behavior Set (12 Techniques). The techniques in this set typically do not have a clear relationship with a specific set of standard API calls. This can result from multiple factors. First, certain behaviors are very generic and can be implemented in many different ways, leading to many possible sequences of API calls. Second, certain behaviors are implemented via API call sequences that have multiple uses. Hence, the presence of a certain API sequence is a weak signal that a certain behavior could be present, and reporting a behavior based on such a sequence alone could lead to many false positives. Third, certain behaviors are "out of scope" for our analysis since they do not lead to any API calls at all. Consider, for example, the *T1564: Hide Artifacts* technique. This behavior is typically implemented by hiding all artifacts associated with malicious behavior by creating computing regions that are isolated from common security instrumentation, such as through the use of virtualization technology.

Discussion. We expect our system to deliver better results for the API-call-based techniques. In fact, this is true, and our results for these 17 techniques show an aggregated precision and recall of 97.33% and 95.78%, respectively. Somewhat more surprisingly, our classifier also delivers good results for the generic behavior techniques, with an aggregated precision and recall of 88.40% and 84.29%, respectively. We explored this finding in more depth and found three main reasons. First, in some cases, the classifier simply (and correctly) learned the different ways to implement a certain behavior. Second, the neural network classifier tends to capture any useful pattern, regardless of whether it is directly semantically relevant to the technique or not. Thus, we sometimes find characteristic sequences of API calls that are not directly responsible for the detected behavior but are used around the code that implements the actual behavior, for example, to prepare or set up certain operations. Finally, there are a few cases where the classifier learns API sequences that are characteristic of a malware family (or a small number of families) that are predominantly responsible for a certain (uncommon) technique. Of course, in this case, the classifier will produce correct results, but for the wrong reason. We discuss this aspect in more detail in Section 4.6.

4.3. Comparison with Existing Work

This section presents a comparative analysis of our system, DEEPCAPA, with two state-of-the-art tools from academia and industry: CAPA [20] and Forecast [19]. To ensure a fair comparison, we translated the capabilities reported by Forecast into the MITRE ATT&CK techniques utilized by our system and CAPA. This translation enabled us to align three behaviors detected by Forecast with three corresponding techniques detected by DEEPCAPA and CAPA, specifically *T1055* (Process Injection), *T1071* (Application Layer Protocol), and *1497* (Virtualization/Sandbox Detection).

We conducted two experiments for this comparative analysis. The first experiment evaluates DEEPCAPA and CAPA using a dataset of 14 samples previously used and reported in the Forecast paper [19]. The second experiment compares the performance of Forecast and CAPA with DEEPCAPA using our dataset. The outcomes of these evaluations are summarized in Table 2A and Table 2B, with a comprehensive discussion and analysis provided below.

Table 2A: Comparing capabilities identified by DEEPCAPA (D), Forecast (F), and CAPA (C) for the 14 samples from the Forecast's evaluation.

Family	T1071			T1055			T1497		
	F	C	D	F	C	D	F	C	D
Rokrat	1	0	1	1	0	1	1	0	1
7honest	0	0	0	1	0	1	0	0	0
Bokbot	1	0	1	1	0	1	0	0	**1**
AcridRain	1	0	1	0	0	0	0	0	0
AthenaGo	0	0	**1**	0	0	0	1	0	1
AdamLocker*	1	0	-	1	0	-	0	0	-
Marap	0	0	**1**	1	0	1	0	0	**1**
Abaddon	0	0	0	0	0	0	0	0	0
ATI	0	0	**1**	0	0	0	1	0	1
TeslaAgent*	1	0	-	1	0	-	0	0	-
Andromeda	1	0	1	1	0	1	0	0	**1**
AvCrypt	1	0	1	0	0	0	1	0	1
Avemaria	1	0	1	1	0	1	0	0	0
Aveo	1	0	1	0	0	**1**	0	0	0
Total	9	0	**10**	**8**	0	7	4	0	**7**

Table 2B: Comparison of precision and recall delivered by DEEPCAPA, Forecast, and CAPA for different techniques.

Tech ID	**Forecast**		**CAPA**		**DEEPCAPA**	
	Prec.	**Recall**	**Prec.**	**Recall**	**Prec.**	**Recall**
T1071	98.03	33.95	**-**	**-**	93.88	93.31
T1055	64.43	60.19	100	2.13	94.89	91.50
T1497	88.19	68.42	100	8.58	94.35	94.48
T1112	**-**	**-**	100	6.13	100.00	97.92
T1027	**-**	**-**	100	49.54	90.12	84.13
T1033	**-**	**-**	100	8.18	94.15	92.56
T1056	**-**	**-**	100	16.26	100.00	96.65
T1082	**-**	**-**	100	31.03	99.96	98.52
T1134	**-**	**-**	100	18.96	100.00	96.43
T1070	**-**	**-**	100	1.63	99.28	98.20
T1057	**-**	**-**	100	11.47	91.40	92.72
T1518	**-**	**-**	100	20.5	97.63	93.42
T1564	**-**	**-**	100	8.26	100.00	84.28
Total	**81.34**	**53.61**	**100**	**12.54**	**97.14**	**94.72**

CAPA. CAPA performs static analysis on malicious executable files to detect their malicious capabilities (behaviors). It identifies behaviors using handcrafted rules that specify different signals, such as API calls, strings, etc. In the first experiment, we found that CAPA could not detect any of the three capabilities in any of the 14 samples from Forecast's evaluation. In the second experiment, we used our own malicious executable files and checked for the 12 MITRE ATT&CK techniques that both DEEPCAPA and CAPA can detect. The results showed that CAPA had a high precision of 100% but a low recall of 12.54% over these 12 techniques. This is because CAPA relies on very specific rules that are tuned to identify particular behaviors but fail to capture different variations of implementations of techniques.

Forecast. Forecast employs symbolic execution to explore code and identifies capabilities based on hand-crafted heuristics, which consider API calls, arguments, and data-flow facts. For the first experiment, we processed the 14 samples used by Forecast in its evaluation using DEEPCAPA. We found that DEEPCAPA could not extract behaviors for two samples (Agent Tesla and AdamLocker) as they are .Net executables. However, for the other 12 samples, DEEPCAPA detected all behaviors detected by Forecast, and additionally detected three more true positives for T1071, one more true positive for T1055, and three more detections for T1497.

In the second experiment, we applied Forecast's capability detection rules on the API call sequences extracted by DEEPCAPA. We chose this approach because DEEPCAPA initiates the analysis of the program from the beginning, while Forecast starts its analysis at a later stage of program execution. Although starting Forecast's analysis from the beginning of the program could also work in principle, it could lead to a path explosion due to symbolic exploration. Consequently, we could not directly run Forecast on our memory snapshots when testing on our dataset. We acknowledge that applying Forecast's capability detection rules to our API call sequences may not perfectly represent Forecast's actual performance. This is because DEEPCAPA only considers API call sequences, whereas Forecast also takes into account data-flow facts. Therefore, Forecast's rules may overestimate the presence of capabilities when applied to our API sequences. As a result, we should interpret the precision numbers (Table 2B) reported for Forecast with a grain of salt. However, the recall results are relevant, as they should represent an upper bound of what we would expect the real Forecast system to detect. Our implementation of Forecast's rules yielded an aggregated precision and recall of 58.03% and 41.41%, respectively. In comparison, DEEPCAPA demonstrated an aggregated precision and recall of 97.14% and 94.72%, respectively. The precision and recall values for DEEPCAPA are slightly different from those reported elsewhere in the paper due to the use of only 13 MITRE ATT&CK techniques for this analysis. As previously mentioned, it would be unfair to draw any conclusions based on the precision results. However, we argue that the recall value is relevant and highlights the problem of false negatives (missed behaviors) because manually-written rules and heuristics are much more narrow in scope than the models learned by DEEPCAPA.

4.4. False Positive Analysis

During our evaluation, we observed that for some malware samples, DEEPCAPA detected dormant capabilities, i.e., those not observed during the execution. The number of samples for each such technique is depicted in Figure 3. The observation that, for some techniques, dynamic labels are incomplete prompted us to conduct a thorough false-positive analysis to identify potential false negatives in our ground-truth labels.

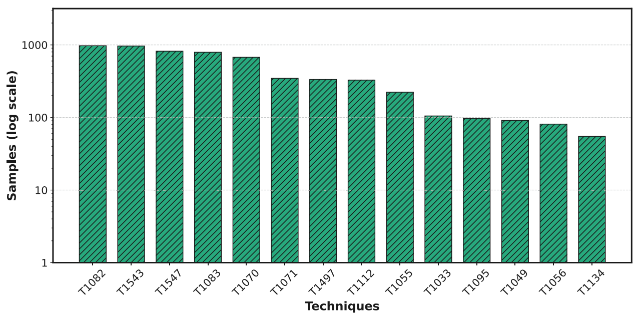

Figure 3: The number of samples with dormant capabilities (per technique) detected by DEEPCAPA.

As discussed in Section 4.1, we derive our ground truth labels by observing malware samples in a sandbox environment and by using static API call heuristics to detect the presence of certain API patterns indicative of a specific technique.

Table 3: Analysis of false positives for API dependent techniques.

TechID	#Examined FPs	#Incorrect FPs	#Validated FPs
T1071	10	6	4
T1055	10	6	4
T1497	10	5	5
T1049	10	4	6
T1518	10	3	7
T1083	4	1	3
T1033	10	1	9
T1082	1	1	0
T1070	10	0	10
T1547	10	0	10
T1057	10	0	10
All	**95**	**27**	**68**

As shown in Table 9 of Appendix A, our system reported 512 API-call-based false positives. For each of these false positives, our classifier reports the presence of a certain technique, but this "detected" technique is not in our ground truth. To determine whether each false positive is indeed a false positive or a true positive (i.e., the ground truth is incorrect), we examined false positives for all techniques in the API-call-based set. Specifically, we randomly selected 10 false positives for each technique and manually reverse-engineered the corresponding malware sample. If a technique contains less than 10 false positives, we analyze all of them. In total, we manually analyzed 95 false positives.

Table 3 shows the results of this analysis. We managed to manually locate the implementation of the detected technique in the sample's code in 27 cases (out of 95 false positives). Extrapolating this finding and assuming a proportional number among all 512 API-call-based false positives, we would get an additional 139 true positives. As a result, the aggregated precision for this set would improve to 98.05% (+0.72%). These results highlight the incompleteness of our ground truth labeling. We believe that our labeling process misses complex, dormant behaviors that our system is capable of capturing. Complex behaviors refer to techniques that can be implemented by different combinations of multiple API calls, including important behaviors such as Process Injection (T1055), Application Layer Protocol (T1071), Virtualization Evasion (T1497), and System Network Connections Discovery (T1049). Our static ground truth labeling only captures narrow sets of API calls and may miss such techniques. Our manual examination supports this hypothesis. For the four aforementioned complex techniques, we found that 52.5% (21 out of 40) of the reported false positives were actually true behaviors missed by our ground truth labels but correctly identified by our system. However, we also identified 68 cases out of 95 false positives that are indeed false positives. A possible reason for these false positives is the insufficient attention mechanism of our model. These validated false positives can occur due to the high noise in the input API call sequence during the training stage. Our system may learn to associate a particular reoccurring noise pattern with a behavior. Then, when a similar noise pattern is encountered in a sample during the testing phase, our model mistakenly identifies the presence of the behavior.

4.5. False Negative Analysis

As shown in Table 9, our system produces 1,477 false negatives. Since our ground-truth labels are based on observations during dynamic execution or based on our conservative API heuristics, a positive label is indeed very likely to be correct. That is, we can assume that all 1,477 reported false negatives are indeed true false negatives. To investigate the root cause for a false negative (x,m), we manually analyzed the disassembly code of x and attempted to locate the implementation of technique m. Specifically, we randomly selected 10 false negatives for each technique and manually reverse-engineered the corresponding malware sample. If a technique contains less than 10 false negatives, we analyzed all of them for a total of 165 false negatives. Table 4 shows the results of our analysis. We noticed two main sources that contributed to the failure of our system to detect relevant techniques.

Indirect API Calls. During our analysis, we found that, in 35 samples, the API calls responsible for a technique are called indirectly. Specifically, malware authors can use the *LoadLibrary* API to load a module into the address space of their running process (by providing the module's path as the argument). Then, they can use the *GetProcAddress* API to fetch the address of the desired function from the export section of the loaded module (by providing the module handle and function name as arguments). Together, these API calls can be used to extract the address of the desired API call defined in the export table of the target module. As the module and API names are passed as arguments to these API calls, our API sequence extraction technique does not find these invocations in memory and thus fails to take capture these API calls. Consequently, our system fails to detect such MITRE ATT&CK technique implementations.

Table 4: Analysis of false negatives for API-call-based techniques.

TechID	#Examined FN	#Indirect	#Training
T1070	10	4	6
T1083	10	2	8
T1082	10	0	10
T1071	10	0	10
T1543	10	8	2
T1497	10	0	10
T1112	10	0	10
T1057	10	1	9
T1055	10	3	7
T1547	10	0	10
T1056	10	4	6
T1033	10	1	9
T1095	10	2	8
T1518	10	0	10
T1134	10	6	4
T1049	10	0	10
T1486	5	0	5
All	**165**	**35**	**130**

Insufficient Training Data. Our model detects the techniques in a sample by using its API calls. However, not all 17 API-call-based techniques are directly implemented by the sample. For example, to modify the registry, the sample may use *RegOpenKey*, *RegCreateKey*, and *RegSetValue* API calls, or it may add a command to the Task Scheduler to execute *reg.exe* or *regedit.exe*. In the latter case, our model cannot find the behavior in the sample itself, although it could detect the API calls for adding a command to the Task Scheduler. However, our training data did not have enough instances of such activities. We found that this was the case for the 130 instances in our dataset, where the samples used indirect methods to implement the MITRE ATT&CK techniques. Some MITRE ATT&CK techniques may have multiple sub-techniques that implement them in different ways. For example, MITRE lists 14 sub-techniques for *T1547: Boot or Logon Autostart Execution* on Windows. Our model may not learn the patterns for all sub-techniques, especially if our training set lacks samples that implement them.

4.6. Attention Analysis

As we explained in Section 3.2, we incorporate attention networks into the design of our system – at the API call sequence level – to address the general lack of explainability that inherently exists with neural networks. In this section, we analyze the attention results for each technique to understand if our classifier captures meaningful sequences of API call that are directly responsible for specific MITRE ATT&CK techniques. For this analysis, we randomly select five samples from the true positives for each of the 17 API call-based techniques. To analyze the attention weights for each sample, we look at the top 10 sequences with the highest weights.

We found that our system worked very well for 12 out of 17 techniques, where the attention weights are assigned to the sequences that directly implement the corresponding behaviors in 58 out of 60 cases. These techniques are *T1071*, *T1055*, *T1497*, *T1112*, *T1486*, *T1082*, *T1070*, *T1083*, *T1134*, *T1543*, *T1056*, and *T1095*.

For example, consider the implementation of technique *T1071 (Application Layer Protocol)*, which is a technique used by adversaries to perform communication and file transfers using the application layer protocol. We found that our system assigns high weights to the sequence consisting of the API calls *socket*, *connect*, *InternetReadFile*, *InternetConnectA*, *WinHttpCloseHandle*, *InternetOpenA*, *HttpOpenRequestW*, and *InternetCloseHandle*. These APIs are a typical way in which an attacker would communicate using the application layer protocol. As another example, technique *T1082 (System Information Discovery)* is reported when a process attempts to get detailed information about the operating system and the underlying hardware. We found that for detecting this technique, our system gives high weights to APIs such as *GetVersion*, *GetVersionExA*, *GetVolumeInformation*, *GetLogicalProcessorInformation*, and *GetVolumeInformationA*. Again, these APIs are the ones that one would expect to see for this technique.

We acknowledge that for the remaining five techniques: *T1547*, *T1057*, *T1033*, *T1518*, and *T1049* the attention weights are not always directly correlated with the technique. However, we note that this does not necessarily mean that our system fails to capture the true signal of the technique. It is possible that there are more relevant sequences that have lower weights but are still sufficient for the classification. Moreover, the attention weights may reflect some patterns of API sequence that are specific to certain malware families, which can also be useful for malware analysis.

4.7. Further Analysis

In the appendix, we present the results of additional experiments to further assess the performance and robustness of DEEPCAPA. In Appendix A, we measure the degradation of our model over time by performing a temporal concept drift analysis. In Appendix B we evaluate the effectiveness of the CFG merging step. Then we compare DEEPCAPA with more basic machine-learning-based approaches that have been effective on related tasks, such as malware detection, in Appendix C. Finally, we conduct an ablation study of our neural network model in Appendix D and show how each component is necessary and beneficial for achieving the best results, as well as how it outperforms simpler baseline approaches.

5. Security Discussion

In Section 4.5, we detailed potential sources of false negatives. Building on that analysis, we examine malware evasion techniques and discuss DEEPCAPA's resilience against such evasion attempts.

DEEPCAPA's design incorporates several features that mitigate the effects of typical malware evasion techniques. *Anti-sandbox techniques*, for instance, are largely ineffective against our approach, as DEEPCAPA operates on full memory snapshots, enabling the analysis of potentially unexecuted (dormant) code. This ability allows us to overcome the limitations often associated with pure dynamic analysis approaches. Similarly, DEEPCAPA demonstrates resilience against *static code obfuscation*. Our graph reduction phase (see Section 3.1) removes non-API call instructions from the CFG, thereby limiting the impact of code obfuscation techniques on our analysis. *Dynamic code generation*, *code-packing*, and *runtime API resolution* also have reduced effects on our system. As detailed in Section 3.1, we can capture dynamically invoked API calls by utilizing multiple snapshots and CFG merging, addressing a key limitation of purely static analysis techniques. While DEEPCAPA effectively counters several evasion methods, we acknowledge that certain techniques may still pose challenges to our analysis. The following paragraphs discuss these techniques, their potential impact, and our proposed countermeasures.

Indirect Jumps. Our system relies on memory snapshots – taken during execution at significant events – to extract code from a malware process. This code is then disassembled and converted into CFGs. After that, we use probabilistic random walks to extract sequences of API calls. To bypass our analysis, an adversary could attempt to hinder CFG construction and/or obfuscate API calls. Our system processes memory snapshots to extract call (target) addresses. The current implementation expects jumps and calls to go to valid addresses. Thus, adversaries could potentially evade our analysis by implementing indirect jumps and resolving the correct address of the control transfer instruction at runtime. For example, as we discussed in Section 4.5, our system has failed to detect certain techniques that leverage indirect API calls (although we note that the number of false negatives due to this limitation is small).

One way to address the issue of indirect control flow transfers is to perform a deeper static analysis. In particular, we could perform data-flow analysis on memory snapshots and look at the arguments of indirect jumps or parameters of specific API calls to handle indirect calls. Alternatively, DEEPCAPA could leverage additional run-time information from the sandbox. In particular, we could dynamically record all jump targets and invocations of API functions, together with their source addresses. While this would only apply to function calls that are actually observed during sandbox execution, our analysis would be able to record and leverage API call information regardless of how the call site is obfuscated (and whether the call is direct or indirect).

We note that indirect jumps are also a potential limitation of related work such as Forecast, which utilizes symbolic execution and data-flow analysis to resolve API call addresses. Forecast may fail to resolve the corresponding API address if an API call is indirectly invoked and the library is not already loaded in memory.

Adversarial samples and API Calls.

DEEPCAPA relies on encoding input samples into lists of API call sequences, which are then processed by a neural network. While neural networks (and, in general, machine learning models) are effective for malware detection, they are also vulnerable to adversarial examples [40]–[43], inputs that are manipulated to avoid detection by a targeted model.

In addition, an adversary might inject "bogus" API calls into legitimate sequences, reordering related API calls, or mixing unrelated calls. Such manipulations can potentially alter the control-flow graph (CFG) by introducing fake nodes (new APIs) or edges (new call links). These changes could impact the effectiveness of our random walk-based sequence extraction, potentially leading to a disproportionate number of misleading sequences. However, it's important to note that excessive manipulation of the API call structure presents a double-edged sword for the attacker. While it may complicate our analysis, it also risks creating unusual CFG patterns that could themselves become indicators of malicious behavior. This inherent trade-off likely constrains the extent to which an adversary can alter the API call structure without inadvertently increasing the sample's detectability.

However, the impact of adversarial samples and API sequences on DEEPCAPA can be significant. They may lead to the misclassification of malicious behaviors or inefficient exploration of the CFG during analysis. To mitigate these risks, we propose incorporating adversarial training techniques [44] into DEEPCAPA. By augmenting our training set with adversarial examples, we can improve the system's robustness against such evasion attempts. Additionally, further research into adaptive CFG exploration strategies and sequence filtering mechanisms could enhance our system's resilience to adversarial manipulations.

Challenges in CFG Generation. Our system is designed to extract Control Flow Graphs (CFG) from memory snapshots using a disassembler, specifically IDA Pro. While our approach is robust, it is not without its challenges. Adversaries can implement techniques that make it difficult for IDA Pro to generate an accurate CFG. One such challenge is the use of exception-handling mechanisms by malware to disrupt the normal flow of execution. This can make it difficult for IDA Pro to generate an accurate CFG as it may not fully understand or be able to predict the behavior of these exceptions. Another significant challenge is the use of polymorphic code by malware. Polymorphic code can change its own instructions while keeping the original algorithm intact, allowing malware to evade signature-based detection. This can pose a significant challenge for IDA Pro, as the changing instructions can prevent it from generating a correct and complete CFG.

Of course, malware analysis remains an arms race, and attackers will continue to introduce techniques that make analysis harder. However, our evaluation demonstrates that our current approach works very well in practice, and we also have room to improve the robustness of our system.

6. Related Work

The field of malware analysis is a rapidly evolving field, with researchers continually proposing novel malware detection or classification methods to keep up with the ever-changing malware landscape [11], [12], [14], [45]–[47]. Despite this progress, the development of post-detection systems that can identify specific malicious capabilities in a given malware program has received less attention. Such systems can significantly aid overworked malware analysts by providing an overview of relevant malware behaviors. Our system targets this post-detection gap by identifying potentially malicious behaviors in the form of techniques defined by the MITRE ATT&CK framework.

In this section, we discuss the related work in this area, categorizing them into two main groups: systems for capability identification and techniques for API extraction. We highlight the strengths and limitations of existing approaches within these categories.

Systems for Capability Identification. CAPA [20], developed by researchers at Mandiant, is a post-detection system that performs static analysis on malicious executable files using manually crafted heuristics to detect their capabilities in the form of MITRE ATT&CK techniques. However, CAPA's approach has limitations, such as its inability to detect capabilities when the malware is packed or its potential lack of generalization due to narrow heuristics. Inspector Gadget [48] isolates specific malware behaviors (techniques) by extracting standalone slices of code using dynamic taint-tracking. These slices are then used to build static models that can identify similar functionality in other programs. However, the models generated by Inspector Gadget are highly specific and require a close match for detection. Forecast [19] is a post-detection framework that processes the memory image of a malicious process to predict capabilities that the malware can execute. It uses a probabilistic model to extract API calls along different execution paths and applies manually crafted heuristics to determine if a certain capability is present in a sample. In contrast to CAPA, Inspector Gadget, and Forecast, our system employs neural networks to automatically learn the features (API call sequences) that characterize different techniques. This makes our approach more general and faster at adapting to new capabilities. SODA [49], MALAF [50], and APILI [51] are systems that map dynamic API call traces to malicious behaviors. However, these tools' dependency on dynamic API call traces to detect behaviors makes them prone to adversarial anti-analysis techniques, thus hindering their effectiveness. On the other hand, our system combines static and dynamic information from memory snapshots, enabling it to process both executed and dormant code even when the malware abruptly terminates.

Techniques for API Extraction. Several projects have used API call sequences to distinguish between malicious and benign activity [21]–[24]. Previous work attempted to extract API call sequences using both static and dynamic analysis. The static extraction of API call sequences involves analyzing the assembly instructions of a binary through static analysis and generating a CFG of the program [52]–[55]. These approaches suffer from the limitation of performing static analysis on malware binaries [56]–[58]. They cannot generate a CFG for packed code, or for code that is generated at runtime and executed in virtual memory. By analyzing multiple process snapshots, DEEPCAPA can generate a CFG for code even if the malware sample was initially packed or if the code was injected into memory at runtime.

Dynamic extraction typically involves executing the malicious code in a sandbox and recording the API call traces [7], [16], [59]–[63]. These API call traces are then used as features for a machine-learning model. However, this approach can only extract API call sequences for executed code, failing to extract dormant sequences. DEEPCAPA overcomes this limitation by analyzing the entire code from process snapshots, allowing it to extract API call sequences from both executed and dormant code.

7. Conclusions

In this work, we propose DEEPCAPA, an automated system that detects high-level, potentially malicious behaviors in the form of MITRE ATT&CK techniques. Our system performs probabilistic random walks on the CFG of multiple memory snapshots of a program as a way to perform robust extraction of a collection of API sequences. These API sequences are our way to model the behaviors of a program. We then leverage deep neural networks and apply them to these API call sequences.

Our results show that DEEPCAPA delivers very promising precision and recall of 95.80% and 93.76%, respectively, across 29 different MITRE techniques. Having compared our system to previous work, we show that our approach to automatically learning behaviors is more general and leads to fewer false negatives.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback. This material is partially based on work supported by the Office of Naval Research under award number N00014-23-1-2387 and by the National Science Foundation under grant no. 2229876, and is also supported in part by funds provided by the National Science Foundation, by the Department of Homeland Security, and by IBM. The opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the ONR, NSF, or any of their federal agency or industry partners.

References

[1] AV-TEST, "File statistics," https://www.av-test.org/en/statistics/malware/, (Accessed: 2022-7-19).

[2] H. Yin, D. Song, M. Egele *et al.*, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 116–127.

[3] C. Kolbitsch, P. M. Comparetti, C. Kruegel *et al.*, "Effective and efficient malware detection at the end host." in *USENIX security symposium*, vol. 4, no. 1, 2009, pp. 351–366.

[4] U. Bayer, P. M. Comparetti, C. Hlauschek *et al.*, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[5] A. Kharaz, S. Arshad, C. Mulliner *et al.*, "{UNVEIL}: A large-scale, automated approach to detecting ransomware," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 757–772.

[6] A.-D. Schmidt, R. Bye, H.-G. Schmidt *et al.*, "Static analysis of executables for collaborative malware detection on android," in *2009 IEEE International Conference on Communications*. IEEE, 2009, pp. 1–5.

[7] C. Li and J. Zheng, "Api call-based malware classification using recurrent neural networks," *Journal of Cyber Security and Mobility*, pp. 617–640, 2021.

[8] G. E. Dahl, J. W. Stokes, L. Deng *et al.*, "Large-scale malware classification using random projections and neural networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 3422–3426.

[9] B. Marais, T. Quertier, and C. Chesneau, "Malware analysis with artificial intelligence and a particular attention on results interpretability," in *International Symposium on Distributed Computing and Artificial Intelligence*. Springer, 2021, pp. 43–55.

[10] S. Jeon and J. Moon, "Malware-detection method with a convolutional recurrent neural network using opcode sequences," *Information Sciences*, vol. 535, pp. 1–15, 2020.

[11] E. Raff, J. Barker, J. Sylvester *et al.*, "Malware detection by eating a whole exe," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[12] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2015, pp. 11–20.

[13] M. Kalash, M. Rochan, N. Mohammed *et al.*, "Malware classification with deep convolutional neural networks," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018, pp. 1–5.

[14] C. Jindal, C. Salls, H. Aghakhani *et al.*, "Neurlux: dynamic malware analysis without feature engineering," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 444–455.

[15] Y. Zhong, H. Yamaki, and H. Takakura, "A malware classification method based on similarity of function structure," in *2012 IEEE/IPSJ 12th International Symposium on Applications and the Internet*. IEEE, 2012, pp. 256–261.

[16] I. Kwon and E. G. Im, "Extracting the representative api call patterns of malware families using recurrent neural network," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, 2017, pp. 202–207.

[17] J. Yan, Y. Qi, and Q. Rao, "Detecting malware with an ensemble method based on deep neural network," *Security and Communication Networks*, vol. 2018, 2018.

[18] S. Ni, Q. Qian, and R. Zhang, "Malware identification using visualization images and deep learning," *Computers & Security*, vol. 77, pp. 871–885, 2018.

[19] O. Alrawi, M. Ike, M. Pruett *et al.*, "Forecasting malware capabilities from cyber attack memory images," in *30th USENIX Security Symposium*, 2021.

[20] "CAPA: Detecting malware capabilities," https://github.com/mandiant/capa/, 2020, (Accessed: 2022-7-19).

[21] S. Forrest, S. Hofmeyr, A. Somayaji *et al.*, "A sense of self for unix processes," in *Proceedings 1996 IEEE Symposium on Security and Privacy*, 1996, pp. 120–128.

[22] D. Mutz, W. Robertson, G. Vigna *et al.*, "Exploiting Execution Context for the Detection of Anomalous System Calls," in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, Gold Coast, Australia, September 2007, pp. 1–20.

[23] R. Canzanese, S. Mancoridis, and M. Kam, "System call-based detection of malicious processes," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 119–124.

[24] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 5–14. [Online]. Available: https://doi.org/10.1145/1287624.1287628

[25] B. E. Strom, A. Applebaum, D. P. Miller *et al.*, "Mitre att&ck: Design and philosophy," *Technical report*, 2018.

[26] H. Rays, "Hex rays official website," https://hex-rays.com/, (Accessed: 2024-5-8).

[27] N. M. Hai, M. Ogawa, and Q. T. Tho, "Obfuscation code localization based on cfg generation of malware," in *International symposium on foundations and practice of security*. Springer, 2015, pp. 229–247.

[28] M. H. Nguyen, D. Le Nguyen, X. M. Nguyen *et al.*, "Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning," *Computers & Security*, vol. 76, pp. 128–155, 2018.

[29] A. Radford, J. Wu, R. Child *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[30] J. Devlin, M.-W. Chang, K. Lee *et al.*, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[31] Y. Liu, M. Ott, N. Goyal *et al.*, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[32] M. Lewis, Y. Liu, N. Goyal *et al.*, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[33] A. Vaswani, N. Shazeer, N. Parmar *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[34] N. Srivastava, G. Hinton, A. Krizhevsky *et al.*, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[35] Virustotal, "Virustotal website," https://www.virustotal.com/gui/home/upload/, (Accessed: 2024-5-8).

[36] S. Sebastián and J. Caballero, "Avclass2: Massive malware tag extraction from av labels," in *Annual Computer Security Applications Conference*, 2020, pp. 42–53.

[37] X. Wu, W. Guo, J. Yan *et al.*, "From grim reality to practical solution: Malware classification in real-world noise," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 2602–2619.

[38] NetworkX, "Graph package," https://networkx.org/, (Accessed: 2023-5-1).

[39] Pytorch, "Tensor library," https://pytorch.org/, (Accessed: 2023-12-1).

[40] I. Rosenberg, A. Shabtai, L. Rokach *et al.*, "Generic black-box end-to-end attack against state of the art api call based malware classifiers," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 490–510.

[41] K. Grosse, N. Papernot, P. Manoharan *et al.*, "Adversarial examples for malware detection," in *European symposium on research in computer security*. Springer, 2017, pp. 62–79.

[42] F. Fadadu, A. Handa, N. Kumar *et al.*, "Evading api call sequence based malware classifiers," in *International Conference on Information and Communications Security*. Springer, 2019, pp. 18–33.

[43] W. Hu and Y. Tan, "Black-box attacks against rnn based malware detection algorithms," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[44] A. Madry, A. Makelov, L. Schmidt *et al.*, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[45] V. S. Sathyanarayan, P. Kohli, and B. Bruhadeshwar, "Signature generation and detection of malware families," in *Australasian Conference on Information Security and Privacy*. Springer, 2008, pp. 336–349.

[46] H. Zhang, W. Zhang, Z. Lv *et al.*, "Maldc: a depth detection method for malware based on behavior chains," *World Wide Web*, vol. 23, no. 2, pp. 991–1010, 2020.

[47] W. Zhong and F. Gu, "A multi-level deep learning system for malware detection," *Expert Systems with Applications*, vol. 133, pp. 151–162, 2019.

[48] C. Kolbitsch, T. Holz, C. Kruegel *et al.*, "Inspector gadget: Automated extraction of proprietary gadgets from malware binaries," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 29–44.

[49] M. S. I. Sajid, J. Wei, B. Abdeen *et al.*, "Soda: A system for cyber deception orchestration and automation," in *Proceedings of the 37th Annual Computer Security Applications Conference*, 2021, pp. 675–689.

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

[51] G.-W. Wong, Y.-T. Huang, Y.-R. Guo *et al.*, "Attention-based api locating for malware techniques," *IEEE Transactions on Information Forensics and Security*, 2023.

[52] Y. Hua, Y. Du, and D. He, "Classifying packed malware represented as control flow graphs using deep graph convolutional neural network," in *2020 International Conference on Computer Engineering and Application (ICCEA)*. IEEE, 2020, pp. 254–258.

[53] J. Yan, G. Yan, and D. Jin, "Classifying malware represented as control flow graphs using deep graph convolutional neural network," in *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2019, pp. 52–63.

[54] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva *et al.*, "Malware detection using assembly and api call sequences," *Journal in computer virology*, vol. 7, no. 2, pp. 107–119, 2011.

[55] O. P. Samantray, S. N. Tripathy, and S. K. Das, "A data mining based malware detection model using distinct api call sequences," *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, vol. 8, no. 7, 2019.

[56] H. Aghakhani, F. Gritti, F. Mecca *et al.*, "When malware is packin'heat; limits of machine learning classifiers based on static analysis features," in *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

[57] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 2007, pp. 421–430.

[58] R. Perdisci, A. Lanzi, and W. Lee, "Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables," in *2008 Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2008, pp. 301–310.

[59] B. Kolosnjaji, A. Zarras, G. Webster *et al.*, "Deep learning for classification of malware system call sequences," in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2016, pp. 137–149.

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

[61] E. Amer and I. Zelinka, "A dynamic windows malware detection and prediction method based on contextual understanding of api call sequence," *Computers & Security*, vol. 92, p. 101760, 2020.

[62] M. Tang and Q. Qian, "Dynamic api call sequence visualisation for malware classification," *IET Information Security*, vol. 13, no. 4, pp. 367–377, 2019.

[63] B. Saha, N. Rani, and S. K. Shukla, "Malxcap: A method for malware capability extraction," in *International Conference on Information Security Practice and Experience*. Springer, 2023, pp. 230–249.

[64] R. Jordaney, K. Sharad, S. K. Dash *et al.*, "Transcend: Detecting concept drift in malware classification models," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 625–642.

Appendix A. Temporal Concept Drift

Concept drift is a well-known problem where the relationship between the input data and the output of a predictive model changes (and diverges). In particular, malware analysis tools are known to suffer from *temporal* concept drift, where performance degrades over *time*, especially as adversaries adapt their implementation to evade detection [64]. Therefore, to keep good performance, it is necessary to evaluate and re-train models regularly over time. As mentioned in Section 4.1, the samples for our (original) training and test datasets were collected in the wild between March 2020 and June 2022. We then obtained more recent samples from the security vendor to carry out a temporal concept drift evaluation. More precisely, we performed this evaluation by testing our model on 3,634 samples, 3,044 samples, and 904 samples found in the wild during July, August, and September 2022, respectively. We first test the performance of DEEPCAPA on the samples for each of these months and observed that our system achieved an accuracy of 84.00%, 86.01%, and 81.34% for the July, August, and September batch, respectively. We then added the samples from July 2022 to our training set and re-tested the newly trained model on samples from August and September 2022. We found that the accuracy for both August and September increased to 88.38% and 85.40%. We then repeated this experiment by adding samples from July and August 2022 to the training set, rebuilt the model, and tested it on the samples from September 2022. The accuracy further increased to 87.71%. Table 5 summarizes the accuracy results for the different experiments.

As expected, our results show the performance (accuracy) improves when we can train on samples that are more recent and "closer" to the time when the test samples are collected. Fortunately, this is a reasonable operational model. That is, in practice, it would be no problem to retrain the model once a month (or even weekly) to take advantage of the most recent data. Moreover, the accuracy does not suddenly collapse; rather, it degrades gradually by one or two percentage points per month.

Table 5: Temporal Concept Drift: We tested DEEPCAPA on samples collected in July, August, and September 2022. We then added July's dataset to our training set and tested our model on August and September's data. Finally, we repeated this process by adding July and August samples to the training set and tested the model on September's data. We found that DEEPCAPA's accuracy indeed gets better as we increase the samples in the training set.

Training Data	July '22	August '22	Sept '22
March '20 - June '22	84.00%	86.01%	81.34%
March '20 - July '22	-	88.38%	85.40%
March '20 - Aug '22	-	-	87.71%

Table 6: Effectiveness of CFG Merging Step.

Snapshot	F1-score	API-call-location pairs
Single Snapshot	82.72%	368
Multiple Snapshots	94.74%	517

Appendix B. Effectiveness of CFG Merging

We employ the CFG Merging technique during the process of API call extraction (Section3.1). This technique enables us to reconstruct the program's CFG from multiple snapshots, which is a critical step for resolving indirect calls and obtaining a comprehensive and accurate representation of the program. To illustrate the significance of the CFG merging step, we conducted an experiment where we extracted API calls from the initial memory snapshots of the samples in the testing set. We then compared the F1 score and the number of unique API-call location pairs with and without the use of the CFG merging step. The results of this experiment are presented in Table 6.

When considering only the first snapshot, we observed an average of 368 unique API-call-location pairs. However, this number increased to an average of 517 when multiple memory snapshots were considered. To evaluate the impact of this reduction in API calls on performance, we conducted capability detection using only the API calls from the initial snapshot, which resulted in an F1 score of 82.72%.

In contrast, DEEPCAPA, achieved an F1 score of 94.74% when it utilized API calls from multiple snapshots. This comparison proves the importance of the CFG merging step in our API call extraction process.

Appendix C. Comparison With N-Gram-Based Classifiers

As we describe in Section 3, our system operates on sequences of API calls. This fact might suggest that simpler classification approaches, for example, those that are based on n-grams extracted over API calls, might achieve comparable results to our neural network. In this section, we perform a comparison to validate if our more complex design choices are actually necessary and beneficial.

For our experiment, we first compute TF-IDF features extracted from 2-grams, 4-grams, and 6-grams over API call sequences. We use these features to train traditional types of machine learning classifiers; in particular, SVMs and random forests. When building individual classifiers per each technique, for each type of classifier, we performed an independent grid search over the hyper-parameters, and we present the best results here. We train and evaluate each classifier with the same training and test sets that are used for training and evaluating our system (as described in Section 4). Note that the training and test split varies per each technique (as we always use a 50%-50% balanced training set).

Our results show that the random forest classifier with 2-grams achieved aggregated precision and recall values of 92.80% and 89.52%, respectively, where as 4-grams features led to aggregated precision and recall values of 91.30% and 87.00%, respectively. For 6-grams, we observed precision and recall values of 90.63% and 85.30%, respectively.

The results for the SVM classifier show aggregated precision and recall values of 93.10% and 90.50%, respectively, when 2-gram features are used. For 4-grams, precision and recall values were 92.70% and 88.12%, respectively, and, for 6-grams, we observed precision and recall values of 91.56 % and 86.58%, respectively.[3]

In general, our results show that DEEPCAPA significantly outperforms both SVM and random forest classifiers that rely on n-gram features, in particular, when looking at the precision results. The overview of the results for this experiment are summarized in Table 7.

Table 7: Comparison between shallow machine learning techniques and the complete DEEPCAPA system. The results show that the DEEPCAPA outperforms the simpler approaches.

Model	Precision (in %)	Recall (in %)
RF-2grams	92.80	89.52
RF-4grams	91.30	87.00
RF-6grams	90.63	85.30
SVM-2grams	93.10	90.50
SVM-4grams	92.70	88.12
SVM-6grams	91.56	86.58
DEEPCAPA	**95.80%**	**93.76%**

Appendix D. Ablation Study

We performed an ablation study to verify two claims about our proposed method DEEPCAPA. First, pretraining on a large corpus of unlabeled API call sequences improves the performance of the classifiers for different MITRE ATT&CK techniques. Second, using a convolutional

3. Overall, we found selecting only the top 85% n-grams (with the highest document frequency) results into a better performance.

neural network (CNN) layer in the fine-tuning stage helps to capture shift-invariant features from the API call sequences. To test these claims, we compare our two-stage training procedure with two baseline models that use different training strategies, while using the same training, validation, and testing sets as used for the fine-tuning stage of DEEPCAPA. The results of this comparison are presented in Table 8.

Table 8: Ablation study results for different training strategies of DEEPCAPA. The results demonstrate that our two-stage training procedure with CNN is effective and necessary for DEEPCAPA.

Model	Precision (in %)	Recall (in %)
Without Pretraining	90.50	91.83
DEEPCAPA with DNN	85.58	83.61
DEEPCAPA	**95.80%**	**93.76%**

For the first baseline, we trained a neural network classifier (both the feature extractor and linear networks) from scratch for each technique, without using any pretraining weights. We use the same neural network architecture as DEEPCAPA and train the model for 30 epochs for each MITRE ATT&CK technique. This approach achieves 90.50% precision and 91.83% recall, which are lower than DEEPCAPA's results. This indicates that our pretraining stage is effective for learning generalizable features that can be leveraged for the fine-tuning tasks.

For the second baseline, we follow the same two-stage training procedure as DEEPCAPA, but we replace the CNN layer with a dense neural network (DNN) layer in the fine-tuning stage. We set the input dimension of the DNN layer to 350×768 (the same as the output dimension of the attention layer) and the output dimension to 64. We fine-tuned this model for 20 epochs for each MITRE ATT&CK technique. In our evaluation, DNN baseline performed considerably worse than DEEPCAPA, achieving 85.58% precision and 83.61% recall. More surprisingly, the DNN baseline also performs worse than the No-Pretrain baseline, despite being initialized with pretraining weights. Thus implying that the CNN layer is indeed effective in capturing shift-invariant features and enhancing the performance of the classifiers.

Appendix E. CFG Exploration Strategy

In this section, we provide a detailed explanation of our CFG exploration strategy. We perform probabilistic random walks over the CFG to extract fixed-length sequences of API calls (in the following paragraphs, the length of each sequence is denoted by A). Probabilistic walks over graphs have previously been used to extract dominant flows in graphs for link prediction and graph classification tasks. We use this approach to traverse the CFG to extract sequences of APIs while giving higher weight to the blocks having a larger number of function/API call instructions.

Prepare probabilistic random walks. The probabilistic approach is based on Markov chains, which describe the probability of transitioning from one state to another using a transition probability matrix. We compute the transition matrix once at the beginning of the analysis, and it is used to select the next block while exploring the CFG.

To calculate the transition matrix for a CFG, we first compute the weight of each block B_i in the graph, denoted by $|B_i|$. This is simply the total number of function/API call instructions in that block. Then, we calculate the adjacency matrix. The adjacency matrix of a CFG with n blocks, which we assume to be ordered from B_1 to B_n, is defined as a $n \times n$ matrix $\mathbb{A}$, in which:

$$\begin{cases} \mathbb{A}_{ij} = 1 & \text{if there exists a path from } B_i \text{ to } B_j, \\ \mathbb{A}_{ij} = 0 & \text{otherwise.} \end{cases} \tag{2}$$

We use $\mathbb{A}$ to calculate the weight matrix $\mathbb{W}$ of the CFG. $\mathbb{W}$ is also a $n \times n$ matrix, in which:

$$\begin{cases} \mathbb{W}_{ij} = \mathbb{A}_{ij} + |B_j| & \text{if } \mathbb{A}_{ij} = 1 \\ \mathbb{W}_{ij} = 0 & \text{otherwise.} \end{cases} \tag{3}$$

We then use $\mathbb{W}$ and $\mathbb{A}$ to calculate the transition matrix $\mathbb{P}$ of the CFG, which is a $n \times n$ matrix that denotes the probability of transitioning from any block in the CFG to any other block:

$$\mathbb{P}_{ij} = \frac{\mathbb{W}_{ij}}{\sum_{t=1}^{n} \mathbb{W}_{it}} \quad \forall i, j. \tag{4}$$

Note that $\sum_{j=1}^{n} \mathbb{P}_{ij}$ is one if and only if there exists a forward edge from B_i, otherwise it is zero.

Algorithm 1: CFG Exploration Strategy.

```
Function Walk(G, B, ℙ, CallList, L, A):
    Input  : G: CFG of the function to explore, B: Block within the
             CFG to explore, ℙ: Transition matrix of CFG, CallList:
             List of call instructions in B, L: Current sequence of
             API calls, A: Target length of the API call sequence
    Output : Sequence L of API calls
    while Len(L) < A do
        foreach Inst in CallList do
            if Len(L)>=A then
                break
            end
            if Inst is a call to function f then
                Gn = GetCFG(f)
                Bn = GetEntryBlock(f)
                ℙn = GetTransitionMatrix(Gn)
                CallList = GetCalls(Gn, B)
                Walk(Gn, Bn,Pn, CallList, L, A)
            end
            if Inst is an invocation of API a then
                L.append(a)
            end
        end
        B = GetNextBlock(ℙ, B)
        if B == NULL then
            break
        end
        else
            CallList = GetCalls(ℙ, B)
        end
    end
```

Perform a probabilistic random walk. Algorithm 1 explains our probabilistic random walk strategy. For each walk, we pick the CFG G of the most unexplored function (initially,

Table 9: Performance of DEEPCAPA. The first set of rows represent API-call-based techniques. The second set of rows are for generic behavior techniques. Note that one malware sample can present multiple techniques.

Tech. ID	Tech. Name	# Test Samples	#TP	#TN	#FP	#FN	Precision (%)	Recall (%)
T1070	Indicator Removal	2652	1801	805	13	33	99.28	98.20
T1083	File and Directory Discovery	2652	2266	347	4	35	99.82	98.48
T1082	System Information Discovery	2652	2335	281	1	35	99.96	98.52
T1071	Standard Application Layer Protocol	2652	1366	1099	89	98	93.88	93.31
T1543	Create or Modify System Process	2652	1810	821	0	21	100.00	98.85
T1497	Virtualization/Sandbox Evasion	2652	1403	1083	84	82	94.35	94.48
T1112	Modify Registry	2590	1268	1295	0	27	100.00	97.92
T1057	Process Discovery	2224	1031	1015	97	81	91.40	92.72
T1055	Process Injection	2070	947	984	51	88	94.89	91.50
T1547	Boot or Logon Autostart Execution	2036	925	917	101	93	90.16	90.86
T1056	Input Capture	1852	895	926	0	31	100.00	96.65
T1033	System Owner/User Discovery	1774	821	836	51	66	94.15	92.56
T1095	Non-Application Layer Protocol	1288	616	644	0	28	100.00	95.65
T1518	Software Discovery	882	412	431	10	29	97.63	93.42
T1134	Access Token Manipulation	840	405	420	0	15	100.00	96.43
T1049	System Network Connections Discovery	336	152	157	11	16	93.25	90.48
T1486	Data Encrypted for Impact	444	217	222	0	5	100.00	97.75
-	**API-Call-Based Set**	**32,287**	**18,670**	**12,283**	**512**	**822**	**97.33%**	**95.78%**
T1036	Masquerading	2652	1455	715	221	261	86.81	84.79
T1027	Obfuscated Files or Information	1084	456	492	50	86	90.12	84.13
T1562	Impair Defenses	850	370	389	36	55	91.13	87.06
T1564	Hide Artifacts	814	343	407	0	64	100.00	84.28
T1047	Windows Management Instrumentation	614	260	265	42	47	86.09	84.69
T1568	Dynamic Resolution	94	40	37	10	7	80.00	85.11
T1485	Data Destruction	280	112	128	12	28	90.32	80.00
T1053	Scheduled Task / Job	108	44	50	4	10	91.67	81.48
T1014	Rootkit	218	89	89	20	20	81.65	81.65
T1203	Exploitation for Client Execution	108	42	51	3	12	93.33	77.78
T1059	Command and Scripting Interpreter	274	107	104	33	30	76.43	78.10
T1552	Unsecured Credentials	460	195	200	30	35	86.67	84.78
-	**Generic Behavior Set**	**7,556**	**3,513**	**2,927**	**461**	**655**	**88.40%**	**84.29%**
-	**All**	**39,843**	**22,183**	**15,210**	**973**	**1,477**	**95.80%**	**93.76%**

this pick is random) and use it as an argument when invoking the function `Walk`. Other arguments to `Walk` include a block B in G as the starting point of the exploration, the transition matrix $\mathbb{P}$ of the CFG, a list *CallList*, which consists of the function and API call instructions in B, an empty list L, which will eventually store the sequence of APIs, and A, the maximum length of the sequence of API calls.

During the exploration of each block, we iterate (Line 3) over the instructions (*Inst*) in its *CallList* and check whether they are (internal) function calls or external API invocations. If the instruction is a function call, we follow the edge in the CFG and continue our walk at the first block of the callee function (Line 7-13). If the instruction is an API invocation, we append the API name to L (Line 14-16).

After iterating over all instructions in a block, we explore the next block in the CFG. To randomly select the next block (in case there are multiple successor nodes), we call *GetNextBlock* (Line 18). The *GetNextBlock* function takes the current block and the transition matrix $\mathbb{P}$ as arguments. If the current block B_i has at least one successor in the CFG (that is, $\sum_{j=1}^{n} \mathbb{P}_{ij} = 1$), *GetNextBlock* returns one of these blocks by performing a weighted random selection. If the current block does not have any successor block (that is, $\sum_{j=1}^{n} \mathbb{P}_{ij} = 0$), the function simply returns *NULL*. We keep exploring all the blocks in G until either the length of L becomes equal to our maximum sequence length A, or we explore all blocks in G. In both cases, `Walk` outputs L.

After exploring G, we check if the length of L is equal to A. If the length is equal to A, we store L and start a new walk, picking again the most unexplored function. If the length of L is less than A, we continue our exploration by calling `Walk`, initialized with a function randomly selected from the set of functions that call G. Basically, we simulate a function return of G and continue the exploration at one of its call sites. If no such caller can be found in the CFG, the random walk stops with a sequence that is shorter than A. At the end of the CFG Exploration stage, we extract S sequences of API invocations, where S is the number of probabilistic random walks performed on the sample's CFG, and each sequence consists of A APIs.