Tarallo: Evading Behavioral Malware Detectors in the Problem Space

Gabriele Digregorio[1], Salvatore Maccarrone[1], Mario D'Onghia[1], Luigi Gallo[2], Michele Carminati[1], Mario Polino[1], and Stefano Zanero[1]

[1] Politecnico di Milano, Milan, Italy
{gabriele.digregorio,mario.donghia,michele.carminati, mario.polino,stefano.zanero}@polimi.it
salvatore.maccarrone@mail.polimi.it
[2] Cybersecurity Lab, TIM S.p.A., Turin, Italy
luigi1.gallo@telecomitalia.it

Abstract Machine learning algorithms can effectively classify malware through dynamic behavior but are susceptible to *adversarial attacks*. Existing attacks, however, often fail to find an effective solution in *both* the feature and problem spaces. This issue arises from not addressing the intrinsic nondeterministic nature of malware, namely executing the same sample multiple times may yield significantly different behaviors. Hence, the perturbations computed for a specific behavior may be ineffective for others observed in subsequent executions. In this paper, we show how an attacker can augment their chance of success by leveraging a new and more efficient feature space algorithm for sequential data, which we have named *Position Sensitive - Fast Gradient Sign Method*, and by adopting two problem space strategies specially tailored to address nondeterminism in the problem space. We implement our novel algorithm and attack strategies in *Tarallo*, an end-to-end adversarial framework that significantly outperforms previous works in both white and black-box scenarios. Our preliminary analysis in a sandboxed environment and against two Recurrent Neural Network (RNN)-based malware detectors, shows that Tarallo achieves a success rate up to 99% on both feature and problem space attacks while significantly minimizing the number of modifications required for misclassification.

Keywords: adversarial machine learning · dynamic analysis · malware detection

1 Introduction

Malware detectors rely on *software analysis techniques* to identify malicious programs. These techniques are grouped into two main categories: *static* and *dynamic*. Static analysis techniques examine the code without running the program, such as strings or IP addresses within the code [20]. While providing a fast way to analyze software, static analysis falls short against obfuscated samples [11,48,16,53]. On the other hand, *dynamic analysis* gathers information while the program is running, capturing its *dynamic behavior*. This includes system calls, network traffic, and memory usage. One commonly employed dynamic feature is the sequence of executed Application Programming Interface (API) calls, which can effectively represent the program functions and goals.

Machine Learning (ML), and particularly Deep Learning (DL), can be employed to detect or classify malware, leveraging both static [38,4,5] and dynamic [39,54,32] features. They represent a more effective alternative to traditional approaches focusing on pattern identification (e.g., byte signatures) or heuristics due to their well-understood generalization capabilities [31,7]. Nonetheless, ML algorithms are susceptible to *adversarial* attacks, which consist in forcing an ML model to misclassify specially *perturbed* samples [18]. These attacks may heavily impact the reliability of ML for cybersecurity applications, forcing their advocates to preemptively research and address vulnerabilities. A substantial body of work in adversarial attacks against static analysis and ML-based malware detectors exists in the literature [24,23,45,27,10]. Fewer works have addressed the security of classifiers employing dynamically extracted features. Of great relevance are the works by Rosenberg et al. [43,42], which introduced an adversarial framework that can target a variety of ML-based malware detectors that employ API call sequences. The authors also discussed and showed how to modify a program's dynamic behavior without *breaking* its functionality, an essential requirement for the attack to be considered successful, formally known as preserved semantics [35]. However, these works do not explicitly discuss or evaluate the real impact of their attacks in the *problem space*. Our preliminary experiments show that only "attacking" the final ML classifier does not suffice to evade the *whole* detection system. This is due to the nondeterminism and probabilistic nature of malware behavior. Hence, computing an adversarial perturbation after observing a specific behavior may not be enough to guarantee a successful attack in subsequent executions. Furthermore, as shown in Section 5, the original feature space attack proposed by Rosenberg et al. [43] requires a very large number of additional API calls to evade newer and more sophisticated classifiers such as [25,54]. This may affect both the stealthiness of the attack as well as its capacity to deliver its original and intended behavior.

Our main contribution consists in designing an Adversarial Machine Learning (AML) attack that is effective both in the feature *and* problem spaces while targeting systems that employ sophisticated DL algorithms such as [25,54]. First, we propose *Position Sensitive - Fast Gradient Sign Method*, a new adversarial attack specific to sequential data, able to produce evasive behaviors with minimal perturbations. Then, we address the problem space limitations through two variants: with the first, the attacker executes several times the malware sample, recording all behaviors, and then selects the longest. The second approach consists of performing the attack while considering multiple behaviors at once.

We preliminary evaluate our attacks through an array of experiments against two state-of-the-art Recurrent Neural Networks (RNNs)-based malware detectors, measuring the attack success in the *problem space* [35] in both white- and black-box scenarios. We confirm that after modification and re-execution in the sandbox, the adversarial samples can still evade detection. Our more stringent criterion contrasts with those adopted in previous works [43,42], which solely focused on evaluating the evasion rate in the feature space. Tarallo achieved a success rate of 99% in the feature space. Additionally, it significantly reduced the number of injections required to achieve misclassification, with an average of 27 additional API calls required, a value significantly lower than what is required by current state-of-the-art approaches against weaker models. In the problem space experiments, our method attains a success rate of up to 99%. While securing high evasion rates, our approach also maintains the original malware functionality. In 89% of the cases, the modified malware preserves its original behavior, thus ensuring its capability to execute its original malicious functions while evading detection. Overall, the results from our experimental evaluations highlight the effectiveness and efficiency of our approach, particularly when compared to state-of-the-art attacks.

The main contributions of this research are summarized below:

- We introduce *Position Sensitive - Fast Gradient Sign Method*, a novel AML attack algorithm that targets discrete sequences, which, against RNN models, proves to be more effective and efficient than state-of-the-art approaches.
- We design two strategies to compute an optimal API call sequence that aims to evade detection in the problem space, addressing the nondeterminism between executions.
- We propose a less invasive code-modification strategy that preserves the original functionality of modified malware samples.
- We present Tarallo, an end-to-end framework that modifies the apparent dynamic behavior of malware, revealing vulnerabilities in state-of-the-art RNN-based detection systems. We make Tarallo publicly available to ensure reproducibility and encourage further research on the topic[3].

2 Related Work

Mimicry Attacks. In the original formulation [50], mimicry attacks aimed to camouflage malicious activities by emulating legitimate goodware behavior. A follow-up work refined this strategy, enabling to fool Intrusion Detection Systems (IDSs) that monitor system calls by adding ones that have no effect or swapping some for equivalent system calls [51]. Nevertheless, mimicry attacks, as well as the related countermeasures (e.g., [17]), targeted IDSs relying on simpler anomaly detection techniques (e.g., by comparing recorded behaviors against known benign ones and modeling behaviors as automata [52,44,46]) than the one based on ML and DL targeted in this work.

DL-based Behavioral Malware Detection. This section presents a brief discussion of DL algorithms for malware detection employing dynamically extracted API call sequences as features. In particular, we focus on the use of RNNs to

[3] https://github.com/necst/Tarallo

process long sequences of API calls, as those are the targets of the attacks presented in this paper. We refer the readers interested in a more in-depth analysis of the topic, including the use of traditional ML as in [49,47,13], to [6].

In [54], the authors presented a feature representation for arguments of API calls, in contrast with most existing works that focus primarily on API names. They designed a Deep Neural Network (DNN) architecture tailored to process these features, which are encoded using a hashing trick method. Samples are executed in a sandbox that records the sequences of API calls. To capture the relationships between these API calls, the model incorporates gated-Convolutional Neural Networks (CNNs), batch normalization layers, a bidirectional Long Short-Term Memory (LSTM) [3], a global max-pooling layer, and a dense layer. This configuration achieved an accuracy of $\approx 95\%$ and a recall of $\approx 71\%$.

In [25], the authors introduced a DNN for dynamic malware detection based on intrinsic features extracted from API call sequences. Similar to [54], each malware sample is executed in a sandbox, where all the API call sequences are recorded. However, instead of using arguments to encapsulate the semantic information of each API call, their proposed model considers three distinct attributes: the category, the action, and the operation object associated with it. Let us consider the API *RegCreateKeyExW*: here, *Create* is the action, *RegKeyEx* the operation object, and *registry* the category. The architecture of the model consists of embedding layers, multi-layer CNNs, a bidirectional LSTM, and dense layers. This architecture achieved an accuracy and recall of 97% and 98%, respectively.

Adversarial Machine Learning. Research in AML has focused on inducing misclassification in ML classifiers at inference time, particularly in the image domain [12,28,18]. However, traditional AML attacks for images (e.g., FGSM [18]) cannot be directly applied to malware detectors due to specific problem-space constraints (i.e., functionality preservation) and the discrete and sequential nature of API call sequences.

There is a growing body of research on perturbing Windows Portable Executable (PE) files; in particular, attacks on static features have been extensively explored in the literature. These attacks involve modifying the headers [41,40] or the raw bytes of PE files [10,23,45,27]. Fewer works have instead targeted dynamic and machine learning-based malware detectors [30,22,21,43,42]. While Ming et al. focused on evading a particular detection strategy based on the *system call dependency graph* [30], we tackle a more general and powerful detection approach that looks at the actual dynamic behaviors (namely, the API call sequence) as discriminating features.

Most closely related to ours are the works by Rosenberg et al. [43,42], which propose an adversarial framework for deceiving ML detectors that work on sequences of dynamically recorded API calls, including RNNs and feed-forward DNNs. Furthermore, they included a tool that modifies malware samples to reflect the required changes in their dynamic behavior. In contrast to [43], our experimental evaluation considers state-of-the-art DL-based behavioral malware detectors. We show that the method from [43] is less effective and efficient against these advanced models, while our strategy proves both highly effective and efficient in evading them. Additionally, Rosenberg et al. [43] assess their attack effectiveness solely in the feature space, deeming it successful if the adversarial algorithm can transform a malicious API call sequence into a variant misclassified as benign by the target detector. In contrast, our study adopts a more complex and realistic approach. An attack is deemed successful if it can alter a malicious sample in a manner that, *upon re-execution*, it produces an API call sequence misclassified as benign by the target model. Our method addresses the significant challenge of varying API call sequences that result from different executions of the same malware sample, an aspect overlooked in [43]. Lastly, in [43] the authors suggest using a wrapper as proxy code, serving as an intermediary between the malware and the DLLs executing the API calls. Our method directly modifies the malware bytecode to intercept API calls. This alteration of the bytecode itself removes the necessity for an extra proxy layer, making the attack more realistic and potentially lowering the likelihood of detecting the modification. Rosenberg et al. also introduced a more efficient black-box attack, necessitating fewer queries to the target model [42]. This method employs a Generative Adversarial Network [9] to mimic genuine benign API call sequences. The authors evaluated the efficacy of this attack with varying query budgets. While it outperforms their earlier method [43] under a limited query budget, their findings also indicate that a gradient-based attack becomes more effective with a larger budget. In contrast, our adversarial algorithm not only surpasses the gradient-based attack from [43] in efficiency and query requirement but also remains effective even with minimal or no knowledge of the target model.

3 Background & Threat Modeling

Adversarial Malware. Based on the work by Pierazzi et al. [35], we formally define the notions employed in this paper. Consider a set of executable programs denoted as E. Let F represent the feature space in which our model operates. We define g as the feature extraction function that maps an executable to a specific point within the feature space, i.e., $g : E \to F$. The readers must notice that g is generally not a surjective function, as it may be generally not possible to identify an executable corresponding to a certain point in the feature space [18]. Let M represent a machine learning classifier, which accepts as input a point in the feature space representing a sequence of API calls. M outputs a prediction from the predefined label set $L = \{malware, goodware\}$, i.e., $M : F \to L$. Given an executable e belonging to the set E and its corresponding representation in the feature space $x_e \in F$, we define the adversarial sample $\hat{x_e}$ for the model M as $\hat{x_e} \triangleq x_e + \delta$ where $\delta \in F$ is the adversarial perturbation, such that M classifies correctly x_e while misclassifies $\hat{x_e}$.

Crafting adversarial samples in the malware problem space presents significant challenges. In contrast to computer vision, where attackers aim to introduce subtle and imperceptible perturbations to images, adversarial malware must be valid and executable programs. This requirement limits the range of possible points in the feature space that can be considered, as not all points correspond to valid and executable programs. Furthermore, the original functionality of the program must be preserved for the attack to be considered successful. This further restricts the extent of possible modifications that an attacker can apply. An attacker can only add extra API calls to the original sequence; in fact, deleting or modifying existing ones would compromise its original functionality.

Threat Model. We examine two different scenarios based on the attacker's knowledge of the target model. In the white-box scenario, the attacker has complete access to its internal parameters. In this case, the model used as the target in the white-box attack directly serves as an oracle. In the black-box scenario, the attacker has no access to the target model architecture or parameters. Instead, the attacker relies on a surrogate oracle, which may have a different structure than the target detector. The only assumption in our experimental evaluation is that the target model employs API call sequences as features.

4 Tarallo

The Tarallo framework operates in an end-to-end manner, to modify the apparent dynamic behavior of malware. It comprises three key components, the first being a Cuckoo sandbox [1] controller that automates the submission of malware samples and the retrieval of the corresponding reports. The second component is responsible for computing the *adversarial* API call sequence and consists of the *Position Sensitive - Fast Gradient Sign Method* (PS-FGSM) algorithm and the two problem space strategies to maximize the evasion probability. Lastly, it comprises a *PE patcher* that parses and modifies a PE file to make it reflect the evasive behavior computed by the second component. Figure 1 depicts Tarallo's workflow. To transform a given malware sample into a functioning adversarial sample, it performs the following steps: 1. It executes the sample multiple times in a controlled environment (sandbox) and records the dynamic behavior of each execution as a sequence of API calls. 2. It runs the PS-FGSM to manipulate the original API call sequence and generate a modified version that can evade the target machine learning model. 3. It modifies the original malware sample to alter its apparent dynamic behavior so that it closely resembles the evasive behavior produced by the adversarial machine learning attack. 4. It executes the modified sample multiple times in the sandbox, recording its dynamic behavior. 5. It lastly evaluates the attack success by examining the effect on the target machine learning model, while also ensuring the preservation of its functionality.

4.1 Position Sensitive - Fast Gradient Sign Method

The main contribution of this work is a novel AML attack designed to target ML models that rely on discrete and sequential data. We call this attack *Position Sensitive - Fast Gradient Sign Method* as it builds upon the *Fast Gradient Sign Method* introduced by Goodfellow et al. [18]. Specific to malware detection, we employ PS-FGSM against models that employ API call sequences as input

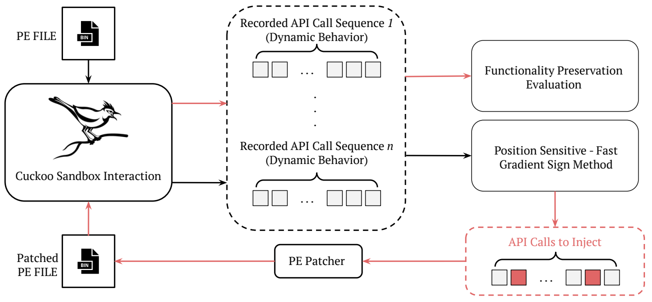
PE FILE
Cuckoo Sandbox Interaction
Patched PE FILE
Recorded API Call Sequence 1 (Dynamic Behavior)
Recorded API Call Sequence n (Dynamic Behavior)
Functionality Preservation Evaluation
Position Sensitive - Fast Gradient Sign Method
API Calls to Inject
PE Patcher

Figure 1. The workflow of the Tarallo framework.

features. The algorithm relies on an oracle with known architecture and parameters. It can either work in a white-box mode, in which the oracle is the target model, or in a full black-box manner. As a variation of the Fast Gradient Sign Method [33,18], the algorithm we present in this work relies on the *sign* of the *inverse* Jacobian of the model's output with respect to its input to minimize the probability of the sample to be classified as belonging to its real class, in this case "malware." As a reminder, the Jacobian matrix represents all the partial derivatives of a function. Since we are considering the *inverse* Jacobian, we are computing the *gradients* of the model from the output to the input.

The PS-FGSM repetitively selects the *best position* within a given subsequence of consecutive API calls (henceforth called *windows*) and adds an extra API call at that position. The algorithm iteratively attacks windows instead of whole sequences because the targeted models are most often RNNs which process one window per time. We take inspiration for the PS-FGSM from the algorithm originally introduced by Rosenberg et al. [43], significantly improving its effectiveness. In particular, we found a way to explicitly compute the position that most affects the output of the model at each iteration, significantly augmenting the impact of each single injection. The rationale behind choosing a specific position within the current window — rather than just focusing on the type of API — is that single API calls cannot be strictly classified as malicious or benign. Therefore, the impact of injecting an API call may vary significantly depending on its position within the original sequence. Indeed, some positions may disrupt "malicious" windows, while others may have little to no effect on the final classification. Furthermore, as the attacker is only allowed to *add* API calls, it intuitively follows that the algorithm must work by displacing a certain API within a sequence, rather than eliminating it.

In Algorithm 1, which showcases the PS-FGSM algorithm for API sequence-based malware detectors, $\perp$ represents the concatenation operation. In its basic implementation, the *stop_condition* is triggered either when the oracle is evaded or when the maximum number of injected API calls is reached. At each iteration, the algorithm processes a window of API calls of size n, which is constrained by the accepted input sequence length of the oracle model. Sequences shorter than n are padded, whereas sequences longer than n are truncated. Moreover, our ap-

Algorithm 1 *Position Sensitive - Fast Gradient Sign Method*

```
Require: $f$ (target model), $x$ (malicious sequence to perturb), $n$ (window size)
 1: procedure AdversarialAttack($f, x, n$)
 2:     $x_{new} \leftarrow [\ ]$
 3:     $r \leftarrow 0$
 4:     for window $w_j$ of size $n$ in $x$ do
 5:         while stop_condition do
 6:             $\mathbf{J_j} \leftarrow J_f(w_j)[f(w_j)]$
 7:             $\mathbf{P}^* \leftarrow$ ComputeBestPosition($\mathbf{J_j}$, $r$)
 8:             $w_j[\mathbf{P}^*] \leftarrow \arg\min_a ||sign(w_j[0:\mathbf{P}^*-1] \perp a \perp w_j[\mathbf{P}^*:n-1]) - sign(\mathbf{J_j})||$
 9:             $r \leftarrow r+1$
10:         end while
11:         $x_{new} \leftarrow x_{new} \perp w_j$
12:     end for
13:     return $x_{new}$
14: end procedure
15: procedure ComputeBestPosition($\mathbf{J_j}$, $r$)
16:     if $r \mod 4 == 0$ then
17:         $norm \leftarrow \|\mathbf{J_j}\|_1$
18:     else
19:         $norm \leftarrow \|\mathbf{J_j}\|_{-1}$
20:     end if
21:     $\mathbf{P}^* \leftarrow$ argmax($norm$)
22:     return $\mathbf{P}^*$
23: end procedure
```

proach includes a mechanism to shift the last API call from the current window to the subsequent one each time a new API call is injected. The position where we inject the additional API call is selected by looking at the norms of the Jacobian matrix. Being API calls discrete entities, these are generally embedded into multidimensional tensors of order k by the classifier. Hence, to evaluate the impact of each API call, we compute their norms along the second-last dimension of their embedding (i.e., assuming k is the number of dimensions of the embedding, we only consider the d_{k-1}^{th} dimension). In this way, the algorithm will produce a matrix of norms with one column for each API call. To select the best candidate in the window, we then select the position with the *greatest smallest norm.* In other words, we collect the smallest norm for each API in that window and then select the greatest. We have empirically verified that this works generally better than selecting the position corresponding to the *greatest absolute norm.* Nonetheless, selecting the *greatest smallest norm* may lead the solution to stagnate; we combine the two norms by adopting the *greatest absolute norm* every c iteration, where c is a small integer (4 in our experiments). Given $U_i \in \mathbb{R}^{d\times n}$ and $U_i := [u_1, \ldots, u_{n-1}, u_n]$ s.t. $u_j = [1,1,\ldots,1]$ iff $j = i$ else $u_j = [0,0,\ldots,0]$, this process is defined as:

$$\mathbf{N^{w,r}} = \sum_{i}^{n} \ell_{\mathbf{1}}\left(\mathbf{J}_{emb\_api_i}\right) * U_i$$

$$\mathbf{P_1^{w,r,*}} := \arg\max_i^n \left(\min \mathbf{N}_i^{w,r}\right), \mathbf{P_2^{w,r,*}} := \arg\max_i^n \left(\max \mathbf{N}_i^{w,r}\right)$$

In the above equations, $\mathbf{N^{w,r}}$ is the norm matrix computed for the $\mathbf{w}^{th}$ API call window at the $\mathbf{r}^{th}$ iteration. $\mathbf{P_1^{w,r,*}}$ and $\mathbf{P_2^{w,r,*}}$ both compute the best position in the $\mathbf{w}^{th}$ window at the $\mathbf{r}^{th}$ iteration, with $\mathbf{P_1^{w,r,*}}$ selecting the one corresponding to the *greatest smallest norm* whereas $\mathbf{P_2^{w,r,*}}$ selects the position where the *greatest absolute norm* is found.

After determining the optimal position, the PS-FGSM selects the API call for injection among those that do not disrupt the original functionality of the program. Moreover, the set is restricted to API calls that are both accurately tracked by Cuckoo and employed as features by the oracle. We denote the ordered set of available API calls as A. Adding API calls not belonging to A would not affect the final classification, as they would be filtered out before classification in the problem space.

Given a window of API calls w and a best position $\mathbf{P}^*$ with respect to w, PS-FGSM selects the API call $a_{inj} \in A$ as:

$$a_{inj} = \arg\min_{a \in A} ||sign(w\,[0 : \mathbf{P}^* - 1] \perp a \perp w\,[\mathbf{P}^* : n - 1]) - sign(\mathbf{J})||$$

In the above equation, we compute the ℓ_1 norm between the sign of the modified window $(w\,[0 : \mathbf{P}^* - 1] \perp a \perp w\,[\mathbf{P}^* : n - 1])$ and the Jacobian of the original one. In other words, we select the API that most closely draws the modified window in the direction of the gradient, expressed by the Jacobian matrix $\mathbf{J}$ (similarly to [43]). Despite the constraints posed by the limited set of API calls — a detail frequently overlooked in previous research [30,22,21,43,42] — Tarallo still manages to evade detection using a particularly small number of injected API calls, especially when compared to Rosenberg et al. [43].

Dealing with nondeterminism. As a premise for this work, we stated that state-of-the-art attacks do not address the nondeterminism typical of real-world malware. This characteristic results in differences among behaviors observed across multiple executions, which in turn strongly impacts the effectiveness of the attack in the problem space. Several factors contribute to nondeterminism, including the system's state at the time of execution (e.g., available resources and network status), operating system scheduling combined with hardware and software multithreading [26], interactions with other executing processes, and evasion techniques like probabilistic control flow approaches [34] and dormant periods [8]. These factors affect the API call sequences observed during different executions of the same malware, a phenomenon that is known in the literature [37] and we encountered in our experiments. While nondeterminism has been studied in the context of dynamic analysis, to the best of our knowledge, this work is the first to address this problem in the context of evading behavioral malware detectors that rely on API call sequences.

We propose two possible strategies to address this issue: Longest Known Behavior (LKB) and Behavior Cascade Optimization (BCO), which maximize the attack success in case the exhibited behavior differs from the one observed by the adversary. Both variations build upon the intuition that the success probability will increase as the confidence score associated with the malicious class decreases. Therefore, the general PS-FGSM is modified to not interrupt the optimization process for the current window when this is classified as benign by the oracle. Instead, the window is optimized until all n_A allowed API calls are injected. The algorithm stores every prediction score obtained after each injection, along with the partial API call sequence composed so far. After all n_A sequences are generated, the algorithm then selects the one that obtained the lowest score by the oracle (i.e., the closest to the goodware label 0). The LKB variant runs the same sample b times, obtaining this way up to b different behaviors. It then selects the behavior that contains the most information, namely the longest sequence of API calls. It lastly performs the variation of the PS-FGSM. Conversely, the BCO builds upon the intuition that the probability of success will increase as the number of behaviors *simultaneously* considered in the feature space grows: i.e., the more behaviors the attacker considers when running the PS-FGSM, the more successful the attack will be. Specifically, the attacker considers b behaviors collected from b executions of the same sample. They sort them out in descending order by looking at the classification scores given by the oracle. Then they attack the most "malicious" sequence and propagate the solution found to all the other behaviors. Suppose the solution found by the PS-FGSM for the first sequence is to inject API calls $\mathbf{a_r}$ before the first a_j occurrence and $\mathbf{a_b}$ before the first a_v, so that $s_1 = |a_o|a_p|a_b|a_j|a_b|a_z|a_v|$ becomes $s_1^* = |a_o|a_p|a_b|\mathbf{a_r}|a_j|a_b|a_z|\mathbf{a_b}|a_v|$. Now suppose $s2 = |a_a|a_b|a_j|a_d|$ and $s_3 = |a_i|a_j|a_k|a_z|a_v|$; before computing s_2^* and s_3^*, the solution found for s_1 is propagated to the two sequences, obtaining in this way: $s_2' = |a_a|a_b|\mathbf{a_r}|a_j|a_d|$ and $s_3' = |a_i|\mathbf{a_r}|a_j|a_k|a_z|\mathbf{a_b}|a_v|$. The attacker then computes the solution $s_2'^*$ from s_2' and propagates the results to s_1^* and s_3'. The algorithm repeats this procedure until all the modified sequences are *simultaneously* classified as benign by the oracle.

4.2 PE Patcher

Lastly, the PE patcher component is responsible for altering the dynamic behavior of malware samples to resemble the adversarial sequence generated by the PS-FGSM. The PE patcher does not require access to the malware source code but is able to preserve its functionality. It supports both x86 and x86-64 PEs. The PE patcher identifies the assembly instructions corresponding to calls to imported APIs through heuristics (e.g., matching call and jump assembly instruction opcodes). It then modifies the arguments of those instructions to redirect them to a code snippet that implements the *hijacking* logic and that was previously injected into the PE. Indeed, our PE patcher directly modifies the malware bytecode, in contrast to state-of-the-art solutions that rely on a wrapper as an intermediary between the malware and the DLLs. This approach makes the PE patcher arguably stealthier. The PE patcher can selectively choose which assembly calls to hijack and modify, while leaving other API calls of the same type unaltered. This level of granularity and flexibility enables to replicate the desired API call sequences with high precision.

Hijacking Logic. When hijacking many API calls, allocating a separate handler for each is inefficient. Such an approach would require modifying extensively the PE, compromising the attack's stealthiness. Instead, the PE patcher creates a single, adaptable code segment capable of handling the different injections. The underlying logic operates following the static directives produced by the PS-FGSM. These directives specify which API calls to hijack and inject. The code segment produced by the PE patcher consists of two modules. The first one is a jump table, which acts as a trampoline for the second module. This second module contains the actual code responsible for performing both the injected and hijacked API calls. Upon hijacking an API call, its assembly call instruction is modified to point to a specific entry in the jump table. Each entry in this table sets up the stack with relevant information about the current execution context, such as the details of the specific API call that has been hijacked. The entire hijacking logic is placed in an additional section added at the end of the PE.

Functionality Preservation. One of the main goals of the Tarallo framework is to preserve the functionality of the malware sample while altering its apparent dynamic behavior. However, formally proving that the modified program behaves exactly as the original is impossible as it can be reduced to the halting problem. Instead, Tarallo adopts a dual strategy.

To avoid changes in its functionality, the PE patcher ensures that each API call performed by the original malware is also executed by the modified one with the same memory and register state. The values of caller-saved registers are stored before calling any injected API, to later enable the reconstruction of the original stack state. In this way, Tarallo ensures that the execution of the original API call produces the intended output, prevents subsequent injected API calls from interfering with one another, and, therefore, does not alter the original functionality. To prevent any disruption in the process flow, the PE patcher ensures that all injected API calls are performed with valid parameters, and conform to the guidelines outlined by the Microsoft documentation [29]. The PE patcher achieves this by passing the correct parameters[4] via the stack or registers, adhering to the calling convention specified by the executable. When the API call needs a pointer as an argument, a pointer within the section added by the PE patcher is provided. If the argument is used only in read mode, the PE patcher fills the location pointed to by that pointer, with null characters or a random string of the required size [29]. Conversely, if the pointer is used in write mode, a pointer to a sufficiently large area of the section, not used for other operations, is provided. Additionally, before executing any injected API call, the stack is aligned to avoid potential issues that could arise from SIMD (Single Instruction, Multiple Data) operations [15].

To demonstrate the preservation of functionality between the original and modified versions, the PE patcher employs an empirical approach. The framework runs both the original and modified versions, recording their API call sequences. It then compares these sequences to determine if the modified version maintains the original behavior. If executed under the same conditions, the sequences should match, except for injected API calls in the modified version. However, replicating identical execution conditions is not always feasible due to nondeterminism. To address this issue, each malware sample is executed multiple times, both before and after the attack. From each run, Tarallo extracts a set of the executed API calls and their parameters. It then computes an invariant that represents the pre-modification behavior. This is done by computing intersections among sets of non-empty API call sequences from the original malware

[4] The list of parameters can be found at: https://github.com/necst/Tarallo/blob/main/ChainFramework/config/config_api_args.py

executions. The invariant serves as a reference point to compare against post-modification behaviors, which are derived through a set union operation of API call sequences from different executions of the modified malware. If the invariant (i.e., the original behavior) is a subset of the post-modification behavior, then Tarallo preserves the program functionality.

5 Experiments

The experiments aim to establish whether the PS-FGSM is effective in deceiving state-of-the-art detectors [25,54] both in the feature and problem spaces, whether the resulting adversarial malware samples preserve their original functionality, and to measure the number of injections required to evade detection (i.e. attack overhead). The feature space evaluation quantifies the attack ability to defeat *only* the DL classifier, which means that the object of the evaluation is solely the API call sequence modified by the PS-FGSM. On the other hand, the problem space evaluation concerns the ability to modify a malware sample that, upon re-execution, will display a behavior able to evade the target DL classifier. In brief, the following experimental evaluation aims to answer four research questions:

RQ1. Is the original functionality of the malware sample preserved after applying the changes to make it evasive? (*Functionality preservation*)

RQ2. Can PS-FGSM generate API call sequences from previously detected malware that can evade state-of-the-art detectors? (*Feature space experiment*).

RQ3. Can PS-FGSM combined with either the LKB or the BCO effectively fool a detection system in an end-to-end manner? (*White- and black-box problem space experiments*).

RQ4. Does the attack transfer to different RNN models employing a different encoding for the API call sequences? (*Black-box problem space experiment*).

Sandbox Setup. We employ 35 Oracle VirtualBox (v5.1.38) virtual machines running Windows 7. Each sample is run with a time limit of 300 seconds [36,16], a duration greater than in [43,42,19]. We disabled the Cuckoo options for simulating random human interactions to ensure a controlled analysis environment and enabled deterministic human interactions, like button-clicking operations, to manage software requiring human interactions before exhibiting their behavior. However, these options cannot encompass all scenarios, leading to software that exhibits limited behaviors in the sandbox environments. This is a recognized challenge in dynamic analysis, part of the Reverse Turing Test[2].

Datasets. Throughout the experiments, we employ different datasets. The first dataset, presented in [25], includes 2000 malware samples first observed in 2019 and 2000 benign executables. The malware was sourced from VirusShare [14], and the benign samples were obtained from popular free software sources like Softonic, SourceForge, and Portableapps. Another dataset, referenced in [54], encompasses the features extracted from 15931 malicious samples detected in April 2017 and an additional 11856 samples from May 2017. This dataset also contains 11417 benign samples gathered in April 2017 and a further 21983 samples collected in May 2017. We also employ Dataset n.375 from VirusShare,

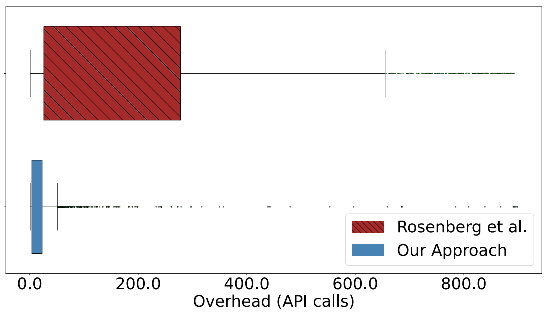

Figure 2. Box plot of the overhead distribution for the feature-level attack against Li et al. [25], using both the proposed PS-FGSM and Rosenberg et al. [43].

comprising 5474 PE files. Of these, 4917 are 32-bit PEs, and 557 are x86-64 PE files. Finally, Dataset n.290 from VirusShare is used, consisting of 7651 PE files. Among these, 7636 are 32-bit and 15 are 64-bit, all collected in May 2017.

Functionality Preservation (RQ1). We executed the original and the modified malware five times in the Cuckoo sandbox. Comparing their behaviors, we observed functionality preservation in 89% of the cases, slightly lower than Rosenberg et al. [43] where all modified malware maintain their functionality. However, 89% is a lower bound. Some differences might result from nondeterminism rather than actual disruptions in functionality, a factor not treated by previous works that do not execute the original and modified malware multiple times. Additionally, Tarallo modifies the malware's bytecode directly instead of using wrappers, increasing realism but also implementation complexity.

5.1 Feature Space Experiment (RQ2)

The feature space experiment evaluates the performance of the PS-FGSM against both state-of-the-art and random algorithms in deceiving the malware detector presented by Li et al. [25]. This evaluation is conducted in a white-box setting, where both the oracle and the target model are the same as Li et al. [25]. Each adversarial sequence is generated from a malicious API call sequence extracted from the execution of a real malware sample. Each test is deemed successful if the adversarial sequence is misclassified as benign by the target detector. Additionally, we measure the number of injected API calls required to achieve misclassification. We refer to this number as the "attack overhead," and we use this as an indicator of the attack efficiency.

Evaluation Result. In this experiment, we employ the VirusShare dataset n.375 and the one used in [25]. The first one is the closest temporally to the one used to train the oracle. Using datasets from different time periods would result in a weak performance by the target detector, which would in turn introduce a bias in favor of the adversarial attack success. We also filter out all the malware samples not detected by the target model, ending up with 2708 malware samples

Table 1. Effectiveness scores for the **feature-level** attack against Li et al. [25] using the proposed PS-FGSM, Rosenberg et al.[43], and a random approach, evaluated with both the VirusShare Dataset No. 375 and the Li et al. Dataset [25].

	VirusShare Dataset n.375			Li et al. Dataset [25]		
Overhead limit	20%	50%	70%	20%	50%	70%
PS-FGSM	**0.9742**	**0.9841**	**0.9863**	**0.9815**	**0.9920**	**0.9925**
Rosenberg et al.[43]	0.6492	0.8556	0.9261	0.5696	0.8178	0.9044
Random Approach	0.2670	0.3674	0.7020	0.2638	0.3634	0.7192

from the VirusShare dataset and 2000 from the one in [25]. In Table 1, we present the results obtained with our PS-FGSM, the approach devised in [43], and an attack consisting in randomly injecting API calls. Both PS-FGSM and the approach in [43] demonstrate almost 100% effectiveness when not imposing strict limits on the attack overhead. However, when we consider a more reasonable attack overhead, the PS-FGSM significantly outperforms the approach in [43]. The box plot in Figure 2 provides a visual representation of the attack overhead by PS-FGSM and the approach in [43]. In the VirusShare dataset, the average overhead for PS-FGSM is 27, indicating that, on average, we have to inject only 27 additional API calls to evade detection. In contrast, the approach in [43] has an average overhead of 202. The same happens for dataset in [25], where the average overhead for the PS-FGSM is 30, while for [43] is 252. Furthermore, when looking at the median overhead, we observe that half of the evaluated samples require 12 or fewer API calls to achieve misclassification when employing the PS-FGSM. In contrast, the median overhead for the approach in [43] is 110, indicating that the majority of samples require a much higher number of injected API calls. Similarly, the median overhead of PS-FGSM is 13 for the dataset in [25], while the approach in [43] has an average overhead of 165.

5.2 White-Box Problem Space Experiment (RQ3)

This experiment shows the capability of Tarallo to produce a modified malware with an evasive dynamic behavior. These settings present additional challenges compared to the feature space attack; e.g., the solution computed in the feature space may include calls to API that are not imported by the original malware. Moreover, it explicitly addresses the nondeterminism in the execution of malware samples in a sandbox, which results in different API call sequences during different executions of the same malware sample, a problem not addressed in previous works such as [43]. To explicitly tackle the complexity of the problem space settings, we employ the LKB and the BCO strategies, both based on the PS-FGSM. We consider the attack successful if and only if re-executing the malicious sample produces an API call sequence that is misclassified as benign by the target model. In the white-box tests, we use the malware detector described by Li et al. in [25] as both the oracle and the target model.

Evaluation Result. The VirusShare dataset n.375 is employed in this experiment; in particular, we keep only executables classified as malicious by the target

Table 2. Longest Known Behavior (LKB) and Behavior Cascade Optimization (BCO) **white-box** attacks overhead and effectiveness in **problem space** against Li et al. [25].

	Overhead Limit	5%			20%			120%
	Arsenal size	(2%, 3%]	(2%, 6%]	Any	(2%, 3%]	(2%, 6%]	Any	Any
LKB	Total Executions	73	165	416	115	267	800	1414
	Evasive Executions	72	129	249	98	189	455	806
	Avg Injected APIs	≈ 19	≈ 19	≈ 22	≈ 56	≈ 56	≈ 74	≈ 406
	Attack Effectiveness	**0.9863**	0.7818	0.5986	0.8522	0.7079	0.5688	0.5700
BCO	Total Executions	90	219	723	100	278	1120	1571
	Evasive Executions	80	196	629	90	252	837	1018
	Avg Injected APIs	≈ 15	≈ 17	≈ 18	≈ 25	≈ 40	≈ 56	≈ 219
	Attack Effectiveness	0.8889	**0.8950**	**0.8700**	**0.9000**	**0.9065**	**0.7473**	**0.6480**

model. Moreover, we exclude all malware with API call sequences shorter than a threshold set to 15 APIs, ensuring that only malware with significant recorded dynamic behavior is considered for modification. For each malware sample, we define an "arsenal" of API calls available for injection, which comprises the intersection of the APIs imported by the malware, the APIs considered safe to inject without disrupting its normal execution, and the APIs recorded by Cuckoo. The results are evaluated using various arsenal size ranges to measure their impact on the attack performance. To evaluate the attack's effectiveness, we run each malware sample five times in Cuckoo. Then, after applying the LKB and BCO strategies, each modified sample is executed five more times per strategy. The resulting API call sequences are submitted for evaluation. In Table 2, we present the results of this evaluation, detailing different arsenal ranges and overhead limits. The arsenal ranges are expressed as the percentage of the total number of API calls recorded by Cuckoo, which is just over 300. For example, a 2% value corresponds to approximately 6 API calls. We compute the attack effectiveness as $EvasiveExecutions/TotalExecutions$.

The problem space constraints have an impact on the attack effectiveness to some extent. However, despite the complex settings, the modified malware still manages to exhibit an evasive apparent dynamic behavior that successfully deceives the white-box target model. For the LKB strategy, we find that the attack effectiveness score decreases as the arsenal size and the overhead limit increase. A larger arsenal size implies a greater variety of API calls available to the malware, which implies that the problem space is broader and, therefore, more challenging from the attacker's perspective, as it amplifies the impact of nondeterminism on the attack performance. As a result, when the malware is re-executed post-attack, the resulting API call sequence may significantly deviate from the original, including differences in API calls used. Moreover, increased overhead generally results in a more fragile attack. Specifically, a high number of injected API calls has an increased chance that variations in the sequence of API calls due to nondeterminism will shift these injections, rendering them ineffective. The BCO strategy shows a more stable attack effectiveness than the LKB strategy, underscoring its superior ability to navigate the breadth of the problem space and handle variations in executed behaviors due to nondeterminism. Moreover, the BCO strategy generally outperforms the LKB strategy, supporting the soundness of its underlying rationale. The only instance where the LKB strategy surpasses BCO is in scenarios with the most limited overhead and arsenal size. This is expected as it is the scenario with a narrower problem space, where nondeterminism has minimal impact. This observation further reinforces the link between the BCO strategy's effectiveness and nondeterminism.

Additionally, we analyze the ability of the BCO and LKB strategies to generate adversarial API call sequences in a constrained environment. We use a subset of the VirusShare dataset no. 375, limiting to a maximum of 800 injected API calls per window and an overall overhead of 120%. Interestingly, the BCO and LKB strategies are successful with different malware samples. In our tests, only about 15% of the samples with an adversarial sequence are targeted by both strategies. Furthermore, the malware samples effectively attacked by the BCO strategy have an average pre-attack sequence length that is roughly 25% shorter than those targeted by the LKB strategy. The LKB strategy succeeds with the most informative (i.e., longest) sequence across different executions of the same malware sample, showing greater effectiveness in creating adversarial sequences with longer available sequences. Conversely, the BCO strategy derives information from all executions of a sample. Consequently, shorter sequences in each run, being less informative, fit within the set overhead limit, making BCO effective in these cases. Notably, unlike the LKB strategy, the BCO strategy's suggested injections are meant for different executions of the same sample, allowing for higher overhead limits compared to LKB. With increased overhead, BCO can attack more samples, including some previously only vulnerable to LKB, although these do not completely overlap.

5.3 Black-Box Problem Space Experiment (RQ3, RQ4)

The black-box, problem space test follows a procedure similar to the white-box one, with a key difference. In this test, the final API call sequences extracted from the modified malware are not evaluated against the model by Li et al. [25], which serves as the oracle for the PS-FGSM attack. Instead, these sequences are tested against a different model, unknown to the attacker. The target model for this black-box test is the DL framework proposed by Zhang et al. [54]. This model presents a significant challenge as it takes into account the arguments of API calls, which are not explicitly manipulated by the AML attack. Instead, the arguments for the injected API calls are defined by the PE patcher to be valid and ensure they do not disrupt the malware's normal execution.

Evaluation Result. We train the black-box model using the dataset and code provided in [54]. We follow the recommendation from the original work to set the prediction threshold corresponding to a FPR of 0.1%. However, we encountered challenges with this configuration as the model is unable to accurately identify the original malware samples from both dataset n.375 and dataset n.290 from VirusShare. This suggests that the chosen threshold and configuration may not be suitable for our specific purposes and dataset. Hence, we select threshold values that correspond to high TPRs, rather than focusing on a specific FPR.

Table 3. Effectiveness scores of Longest Known Behavior (LKB) and Behavior Cascade Optimization (BCO) approaches in **black-box**, **problem space** settings against Zhang et al.[54]—evaluated across different prediction thresholds, True Positive Rate (TPR), and False Positive Rate (FPR)—using the model by Li et al.[25] as an oracle.

Threshold	TPR	FPR	Available Executions		Evading Executions		Attack Effectiveness	
			LKB	BCO	LKB	BCO	LKB	BCO
0.9683	0.992	0.325	309	161	154	82	0.4984	**0.5093**
0.9776	0.989	0.300	263	146	130	71	**0.4943**	0.4863
0.9868	0.986	0.275	225	131	115	75	0.5111	**0.5725**
0.9951	0.978	0.225	162	86	79	51	0.4877	**0.5930**
0.9981	0.971	0.178	141	82	69	52	0.4894	**0.6341**
0.9988	0.968	0.150	132	68	67	39	0.5076	**0.5735**
0.9996	0.958	0.100	115	57	54	33	0.4696	**0.5789**

Although this strategy results in higher FPRs, it allows us to assess the effectiveness of the Tarallo framework in fooling robust machine learning systems. By testing our method with different threshold values, we can demonstrate how this parameter influences the effectiveness of our attack and provide a comprehensive validation of the experimental evaluation. In our tests, we run the original malware samples from the VirusShare dataset n.375 through the Cuckoo sandbox, recording their API call sequences. We repeat this process five times to collect a sufficient number of executions. For each threshold value considered during the test, we select only the malware samples that are classified as malicious by the black-box model in *all* the executions. We attack each sample with both the LKB and BCO attack strategies and then run the modified versions of these malware executables through the Cuckoo sandbox five times for each strategy, recording their evading API call sequences. The results of these experiments are summarized in Table 3. The attack effectiveness is instead computed as $EvadingExecutions/TotalExecutions$. Restricting the target model to a lower FPR —though still above the value reported in the original work— by increasing the classification threshold, leads to fewer correctly classified malware samples (i.e., a lower TPR), particularly those the model is most confident about. This makes attacks on these samples harder for both the LKB and BCO, reducing the effectiveness score compared to other threshold settings. Lowering the threshold results in an increase in the FPR, which may be less realistic, but it also enhances the TPR. This improvement boosts the model's malware detection capabilities, more accurately mirroring real-world conditions. Under these conditions, both attacks tend to exhibit improved performance. However, when we further increase both the TPR and the FPR, the attack effectiveness score starts to decline again. This decrease occurs because the model begins to incorrectly identify a large number of benign behaviors as malicious, leading to a misclassification of the injection in the original API call sequence as harmful. The tests underscore the need to balance TPR and FPR in detection system design, requiring careful domain-specific consideration to optimize accuracy while reducing evasion attack risks. Additionally, the consistently better performance of the BCO strategy confirms that considering multiple sequences during the attack results in more resilient and general adversarial sequences.

6 Conclusions

In this paper, we introduced Tarallo, an end-to-end framework designed to modify the apparent dynamic behavior of malware to deceive ML detectors that employ API call sequences as features. Our first major contribution is a novel AML attack that targets discrete and sequential data. Moreover, we explicitly addressed the problem of nondeterministic execution in a sandbox, which makes the design of problem space attacks particularly challenging. Lastly, we introduced a new approach for modifying the run-time behavior of malicious software that does not require access to the source code. The preliminary experimental evaluation in a sandboxed environment and against two RNN-based malware detectors, showed that our novel AML algorithm, namely the PS-FGSM, outperforms current state-of-the-art algorithms in terms of both effectiveness and efficiency. In both feature and problem spaces, our algorithm achieves up to 99% effectiveness, while also minimizing overhead as measured by the number of injected APIs. Although we showed that obfuscating the behavior of existing malware is a nontrivial problem and, therefore, that current antimalware solutions are very robust, we also demonstrated that such attacks are in fact possible.

Future work will focus on assessing the effectiveness of Tarallo against different malware detectors, especially those not based on RNN. Moreover, we will explore methodologies that allow defenders to remove non-operational APIs from API call sequences, enhancing the overall performance of the detection systems. We also intend to employ the PS-FGSM to build an adversarial training module to strengthen state-of-the-art DL-based detectors.

Acknowledgements This study was carried out within the MICS (Made in Italy – Circular and Sustainable) Extended Partnership and received funding from Next-Generation EU (Italian PNRR – M4 C2, Invest 1.3 – D.D. 1551.11-10-2022, PE00000004). CUP MICS D43C22003120001. Mario D'Onghia acknowledges support from TIM S.p.A. through the PhD scholarship.

References

1. Cuckoo (2024), https://github.com/cuckoosandbox/cuckoo
2. Afianian, A., Niksefat, S., Sadeghiyan, B., Baptiste, D.: Malware dynamic analysis evasion techniques: A survey. ACM Comput. Surv. **52**(6) (nov 2019). https://doi.org/10.1145/3365001, https://doi.org/10.1145/3365001
3. Agrawal, R., Stokes, J.W., Marinescu, M., Selvaraj, K.: Neural sequential malware detection with parameters. In: 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). pp. 2656–2660. IEEE (2018)
4. Anderson, H.S., Roth, P.: Ember: an open dataset for training static pe malware machine learning models. arXiv preprint arXiv:1804.04637 (2018)
5. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: Effective and explainable detection of android malware in your pocket. In: Ndss. vol. 14, pp. 23–26 (2014)
6. Berman, D.S., Buczak, A.L., Chavis, J.S., Corbett, C.L.: A survey of deep learning methods for cyber security. Information **10**(4), 122 (2019)

7. Catak, F.O., Yazı, A.F., Elezaj, O., Ahmed, J.: Deep learning based sequential model for malware analysis using windows exe api calls. PeerJ Computer Science **6**, e285 (2020)
8. Comparetti, P.M., Salvaneschi, G., Kirda, E., Kolbitsch, C., Kruegel, C., Zanero, S.: Identifying dormant functionality in malware programs. In: 2010 IEEE Symposium on Security and Privacy. pp. 61–76. IEEE (2010)
9. Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., Bharath, A.A.: Generative adversarial networks: An overview. IEEE signal processing magazine **35**(1), 53–65 (2018)
10. D'Onghia, M., Di Cesare, F., Gallo, L., Carminati, M., Polino, M., Zanero, S.: Lookin'out my backdoor! investigating backdooring attacks against dl-driven malware detectors. In: Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security. pp. 209–220 (2023)
11. D'Onghia, M., Salvadore, M., Nespoli, B.M., Carminati, M., Polino, M., Zanero, S.: Apícula: Static detection of api calls in generic streams of bytes. Computers & Security **119**, 102775 (2022)
12. Eykholt, K., Evtimov, I., Fernandes, E., Li, B., Rahmati, A., Xiao, C., Prakash, A., Kohno, T., Song, D.: Robust physical-world attacks on deep learning visual classification. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 1625–1634 (2018)
13. Fang, Y., Yu, B., Tang, Y., Liu, L., Lu, Z., Wang, Y., Yang, Q.: A new malware classification approach based on malware dynamic analysis. In: Information Security and Privacy: 22nd Australasian Conference, ACISP 2017, Auckland, New Zealand, July 3–5, 2017, Proceedings, Part II 22. pp. 173–189. Springer (2017)
14. Forensics, C.: Virusshare (2023), http://virusshare.com/
15. Furht, B. (ed.): SIMD (Single Instruction Multiple Data Processing), pp. 817–819. Springer US, Boston, MA (2008). https://doi.org/10.1007/978 $- 0 - 387 - 78414 - 4_220$, $https : //doi.org/10.1007/978 - 0 - 387 - 78414 - 4_220$
16. Galloro, N., Polino, M., Carminati, M., Continella, A., Zanero, S.: A systematical and longitudinal study of evasive behaviors in windows malware. Computers & Security **113**, 102550 (2022)
17. Giffin, J.T., Jha, S., Miller, B.P.: Automated discovery of mimicry attacks. In: Recent Advances in Intrusion Detection: 9th International Symposium, RAID 2006 Hamburg, Germany, September 20-22, 2006 Proceedings 9. pp. 41–60. Springer (2006)
18. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572 (2014)
19. Hariom, Handa, A., Kumar, N., Kumar Shukla, S.: Adversaries strike hard: Adversarial attacks against malware classifiers using dynamic api calls as features. In: Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings 5. pp. 20–37. Springer (2021)
20. Hassen, M., Carvalho, M.M., Chan, P.K.: Malware classification using static analysis based features. In: 2017 IEEE Symposium Series on Computational Intelligence (SSCI). pp. 1–7. IEEE (2017)
21. Hu, W., Tan, Y.: Black-box attacks against rnn based malware detection algorithms. arXiv preprint arXiv:1705.08131 (2017)
22. Hu, W., Tan, Y.: Generating adversarial malware examples for black-box attacks based on gan. In: Data Mining and Big Data: 7th International Conference, DMBD 2022, Beijing, China, November 21–24, 2022, Proceedings, Part II. pp. 409–423. Springer (2023)

23. Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C., Roli, F.: Adversarial malware binaries: Evading deep learning for malware detection in executables. In: 2018 26th European signal processing conference (EUSIPCO). pp. 533–537. IEEE (2018)
24. Kreuk, F., Barak, A., Aviv-Reuven, S., Baruch, M., Pinkas, B., Keshet, J.: Deceiving end-to-end deep learning malware detectors using adversarial examples. arXiv preprint arXiv:1802.04528 (2018)
25. Li, C., Lv, Q., Li, N., Wang, Y., Sun, D., Qiao, Y.: A novel deep framework for dynamic malware detection based on api sequence intrinsic features. Computers & Security **116**, 102686 (2022)
26. Liu, T., Curtsinger, C., Berger, E.D.: Dthreads: efficient deterministic multithreading. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. pp. 327–336 (2011)
27. Lucas, K., Sharif, M., Bauer, L., Reiter, M.K., Shintre, S.: Malware makeover: Breaking ml-based static analysis by modifying executable bytes. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. pp. 744–758 (2021)
28. Machado, G.R., Silva, E., Goldschmidt, R.R.: Adversarial machine learning in image classification: A survey toward the defender's perspective. ACM Computing Surveys (CSUR) **55**(1), 1–38 (2021)
29. Microsoft: Programming reference for the win32 api (2024), https://learn.microsoft.com/en-us/windows/win32/api/
30. Ming, J., Xin, Z., Lan, P., Wu, D., Liu, P., Mao, B.: Impeding behavior-based malware analysis via replacement attacks to malware specifications. Journal of Computer Virology and Hacking Techniques **13**, 193–207 (2017)
31. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Twenty-third annual computer security applications conference (ACSAC 2007). pp. 421–430. IEEE (2007)
32. Or-Meir, O., Nissim, N., Elovici, Y., Rokach, L.: Dynamic malware analysis in the modern era—a state of the art survey. ACM Computing Surveys (CSUR) **52**(5), 1–48 (2019)
33. Papernot, N., McDaniel, P., Swami, A., Harang, R.: Crafting adversarial input sequences for recurrent neural networks. In: MILCOM 2016-2016 IEEE Military Communications Conference. pp. 49–54. IEEE (2016)
34. Pawlowski, A., Contag, M., Holz, T.: Probfuscation: an obfuscation approach using probabilistic control flows. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13. pp. 165–185. Springer (2016)
35. Pierazzi, F., Pendlebury, F., Cortellazzi, J., Cavallaro, L.: Intriguing properties of adversarial ml attacks in the problem space. In: 2020 IEEE symposium on security and privacy (SP). pp. 1332–1349. IEEE (2020)
36. Polino, M., Continella, A., Mariani, S., D'Alessio, S., Fontana, L., Gritti, F., Zanero, S.: Measuring and defeating anti-instrumentation-equipped malware. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14. pp. 73–96. Springer (2017)
37. Polino, M., Scorti, A., Maggi, F., Zanero, S.: Jackdaw: Towards automatic reverse engineering of large datasets of binaries. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings 12. pp. 121–143. Springer (2015)

38. Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.K.: Malware detection by eating a whole exe. In: Workshops at the thirty-second AAAI conference on artificial intelligence (2018)
39. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. Journal of computer security **19**(4), 639–668 (2011)
40. Rosenberg, I., Meir, S.: Bypassing ngav for fun and pro t (2020)
41. Rosenberg, I., Meir, S., Berrebi, J., Gordon, I., Sicard, G., David, E.O.: Generating end-to-end adversarial examples for malware classifiers using explainability. In: 2020 international joint conference on neural networks (IJCNN). pp. 1–10. IEEE (2020)
42. Rosenberg, I., Shabtai, A., Elovici, Y., Rokach, L.: Query-efficient black-box attack against sequence-based malware classifiers. In: Annual Computer Security Applications Conference. pp. 611–626 (2020)
43. Rosenberg, I., Shabtai, A., Rokach, L., Elovici, Y.: Generic black-box end-to-end attack against state of the art api call based malware classifiers. In: Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21. pp. 490–510. Springer (2018)
44. Somayaji, A., Forrest, S.: Automated response using {System-Call} delay. In: 9th USENIX security symposium (USENIX security 00) (2000)
45. Suciu, O., Coull, S.E., Johns, J.: Exploring adversarial examples in malware detection. In: 2019 IEEE Security and Privacy Workshops (SPW). IEEE (2019)
46. Tan, K., Maxion, R.: "why 6?" defining the operational limits of stide, an anomaly-based intrusion detector. In: Proceedings 2002 IEEE Symposium on Security and Privacy. pp. 188–201 (2002). https://doi.org/10.1109/SECPRI.2002.1004371
47. Tian, R., Islam, R., Batten, L., Versteeg, S.: Differentiating malware from cleanware using behavioural analysis. In: 2010 5th international conference on malicious and unwanted software. pp. 23–30. Ieee (2010)
48. Ucci, D., Aniello, L., Baldoni, R.: Survey of machine learning techniques for malware analysis. Computers & Security **81**, 123–147 (2019)
49. Uppal, D., Sinha, R., Mehra, V., Jain, V.: Malware detection and classification based on extraction of api sequences. In: 2014 International conference on advances in computing, communications and informatics (ICACCI). IEEE (2014)
50. Wagner, D., Dean, R.: Intrusion detection via static analysis. In: Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001. pp. 156–168. IEEE (2000)
51. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proceedings of the 9th ACM Conference on Computer and Communications Security. pp. 255–264 (2002)
52. Warrender, C., Forrest, S., Pearlmutter, B.: Detecting intrusions using system calls: Alternative data models. In: Proceedings of the 1999 IEEE symposium on security and privacy (Cat. No. 99CB36344). pp. 133–145. IEEE (1999)
53. You, I., Yim, K.: Malware obfuscation techniques: A brief survey. In: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications. pp. 297–300 (2010). https://doi.org/10.1109/BWCCA.2010.85
54. Zhang, Z., Qi, P., Wang, W.: Dynamic malware analysis with feature engineering and feature learning. In: Proceedings of the AAAI conference on artificial intelligence. vol. 34, pp. 1210–1217 (2020)